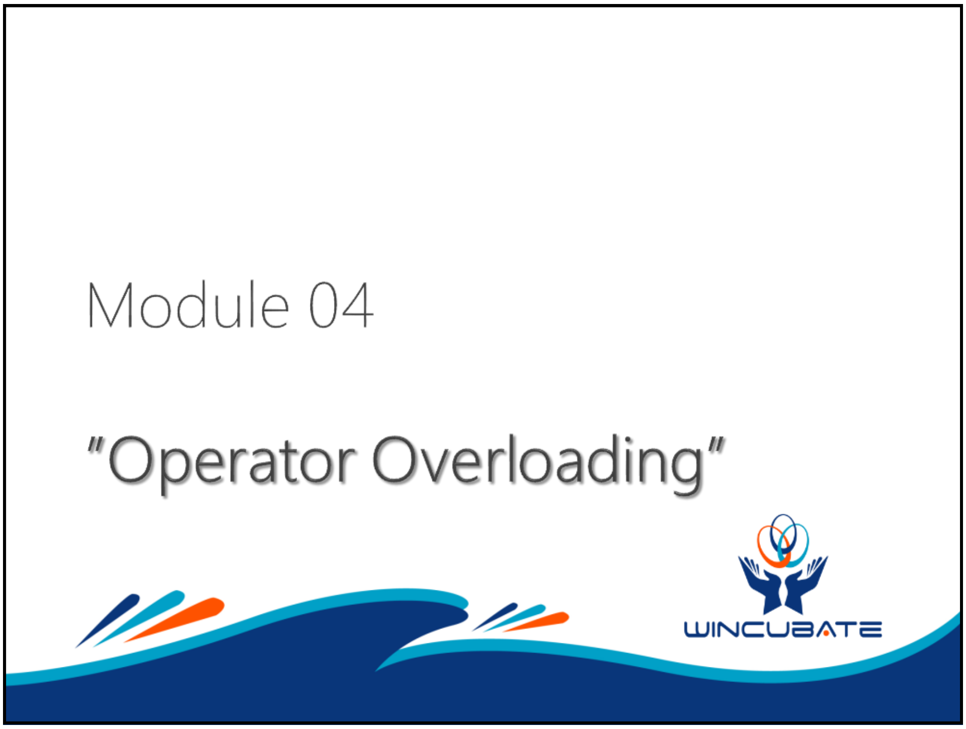
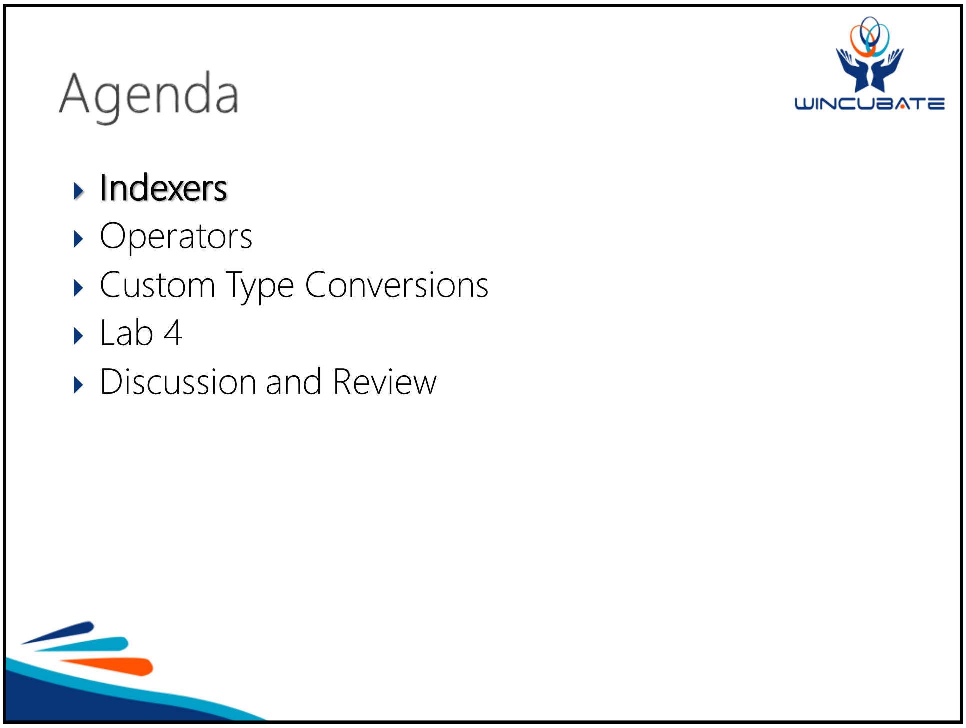


Module 04

"Operator Overloading"

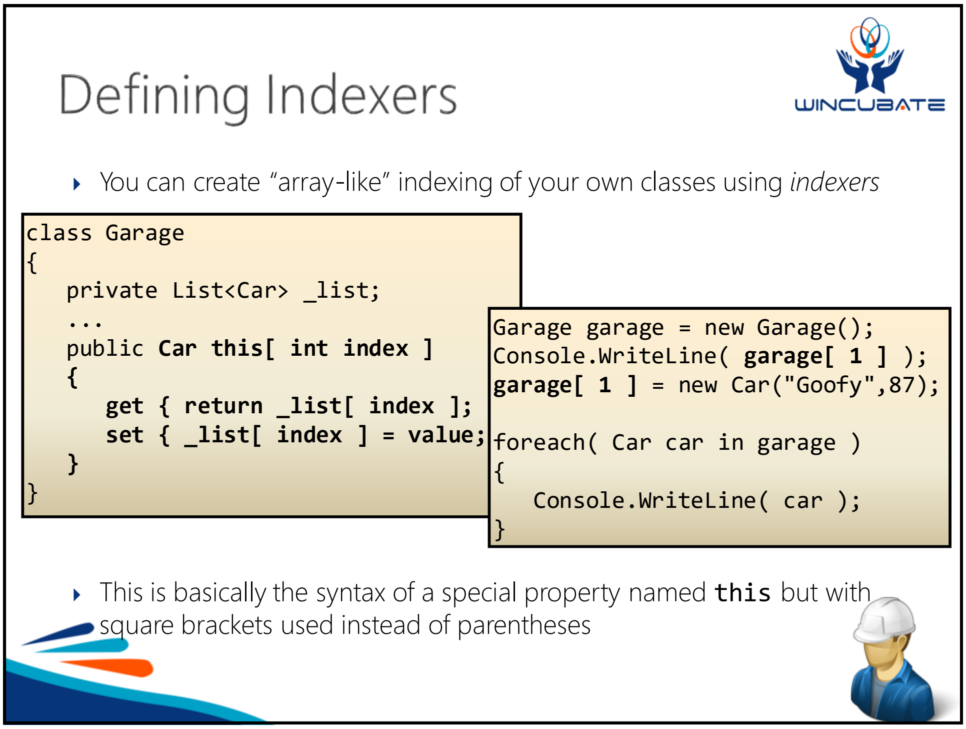
WINCUBATE



Agenda

WINCUBATE

- Indexers
- Operators
- Custom Type Conversions
- Lab 4
- Discussion and Review



Defining Indexers

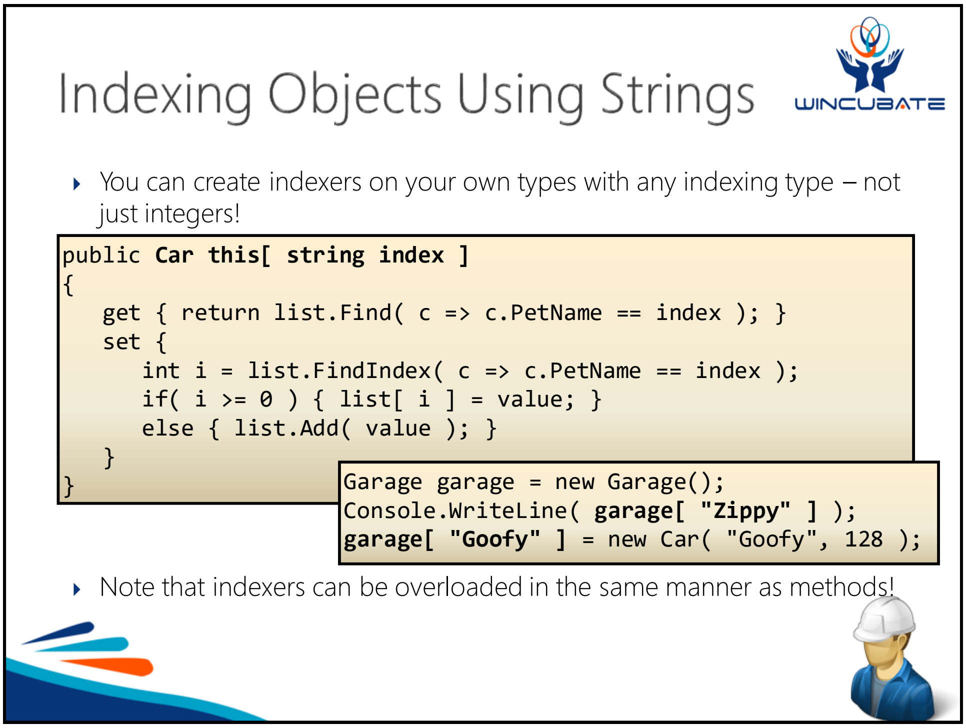
- You can create “array-like” indexing of your own classes using *indexers*

```
class Garage
{
   private List<Car> _list;
   ...
   public Car this[ int index ]
   {
      get { return _list[ index ]; }
      set { _list[ index ] = value;
   }
}
```

```
Garage garage = new Garage();
Console.WriteLine( garage[ 1 ] );
garage[ 1 ] = new Car("Goofy",87);

foreach( Car car in garage )
{
   Console.WriteLine( car );
}
```

- This is basically the syntax of a special property named **this** but with square brackets used instead of parentheses



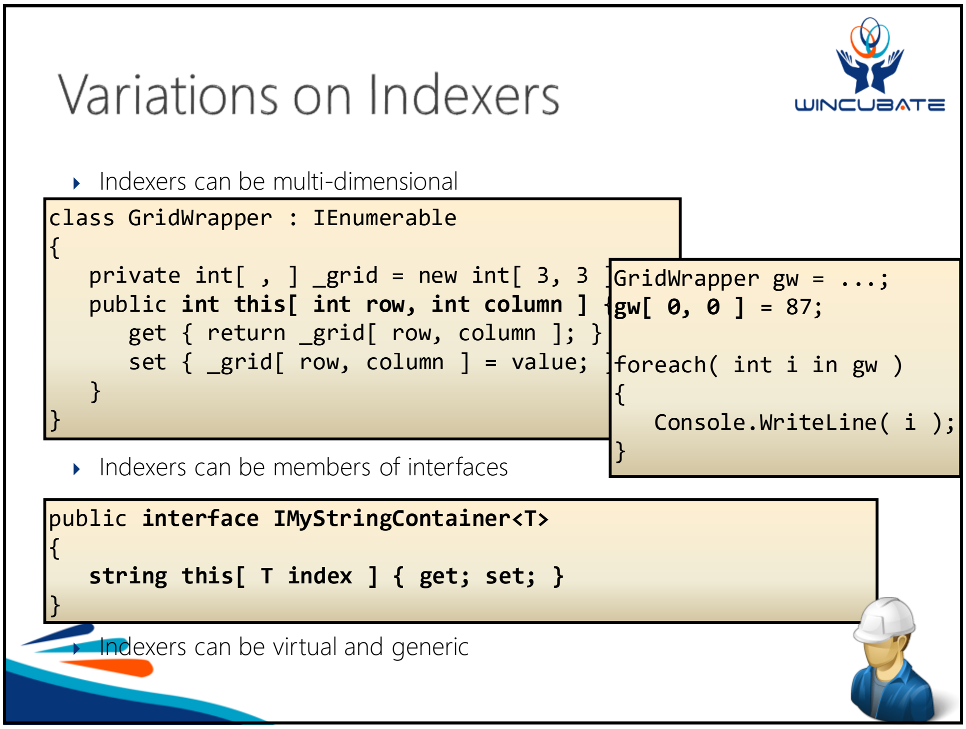
Indexing Objects Using Strings

- You can create indexers on your own types with any indexing type – not just integers!

```
public Car this[ string index ]
{
   get { return list.Find( c => c.PetName == index ); }
   set {
      int i = list.FindIndex( c => c.PetName == index );
      if( i >= 0 ) { list[ i ] = value; }
      else { list.Add( value ); }
   }
}
```

```
Garage garage = new Garage();
Console.WriteLine( garage[ "Zippy" ] );
garage[ "Goofy" ] = new Car( "Goofy", 128 );
```

- Note that indexers can be overloaded in the same manner as methods!



Variations on Indexers

- Indexers can be multi-dimensional

```
class GridWrapper : IEnumerable
{
   private int[ , ] _grid = new int[ 3, 3 ]
   public int this[ int row, int column ] {
      get { return _grid[ row, column ]; }
      set { _grid[ row, column ] = value; ]
   }
}
```

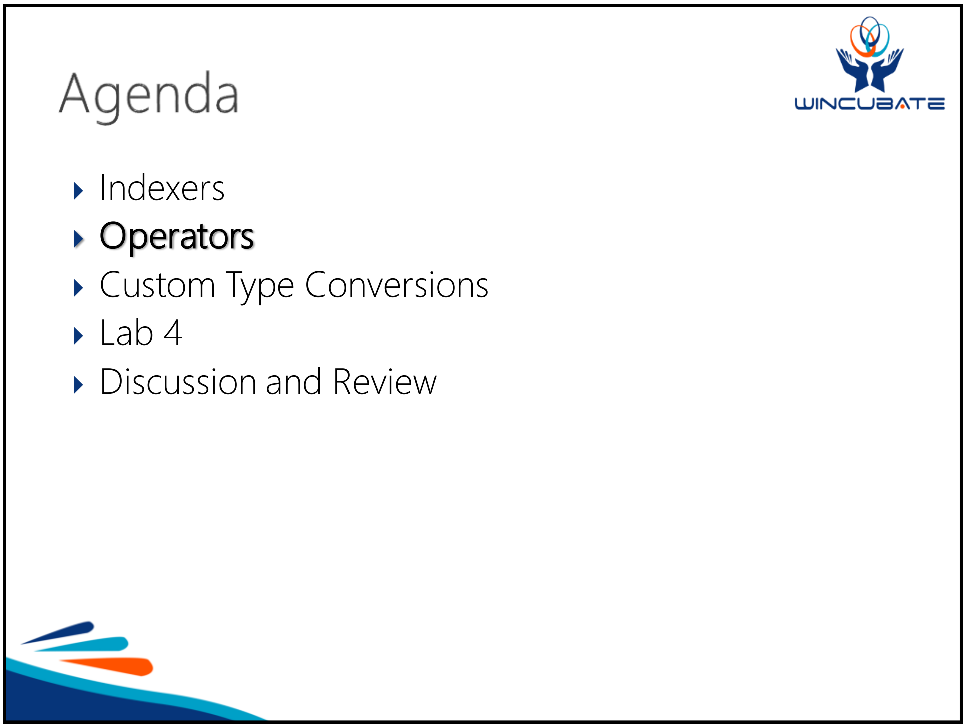
```
GridWrapper gw = ...;
gw[ 0, 0 ] = 87;

foreach( int i in gw )
{
   Console.WriteLine( i );
}
```

- Indexers can be members of interfaces

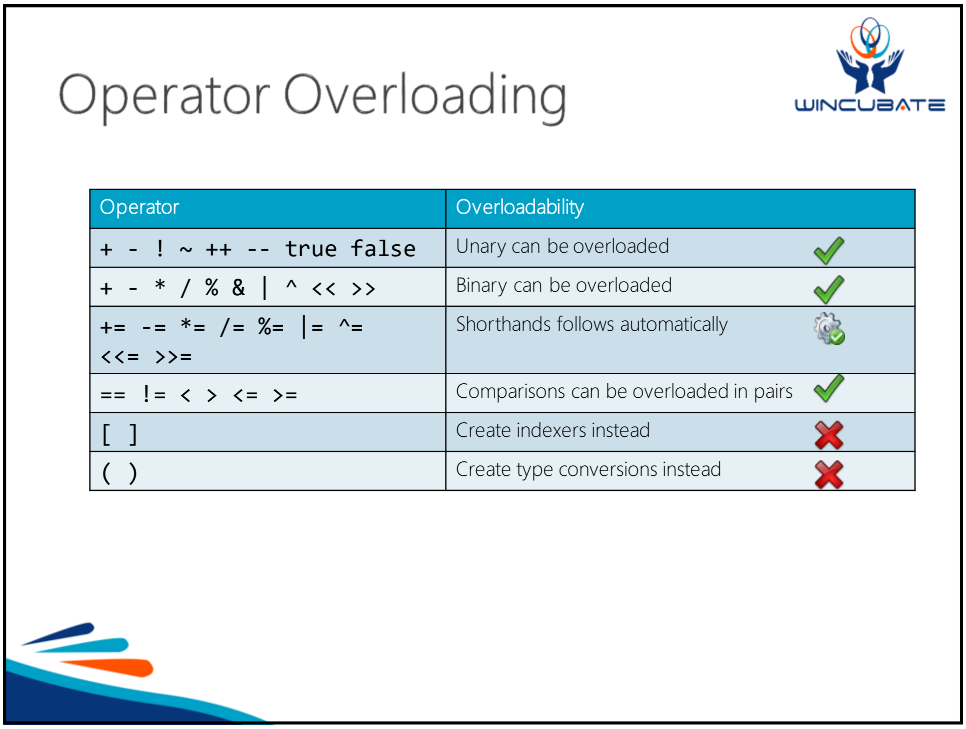
```
public interface IMyStringContainer<T>
{
   string this[ T index ] { get; set; }
}
```

- Indexers can be virtual and generic



Agenda

- Indexers
- **Operators**
- Custom Type Conversions
- Lab 4
- Discussion and Review

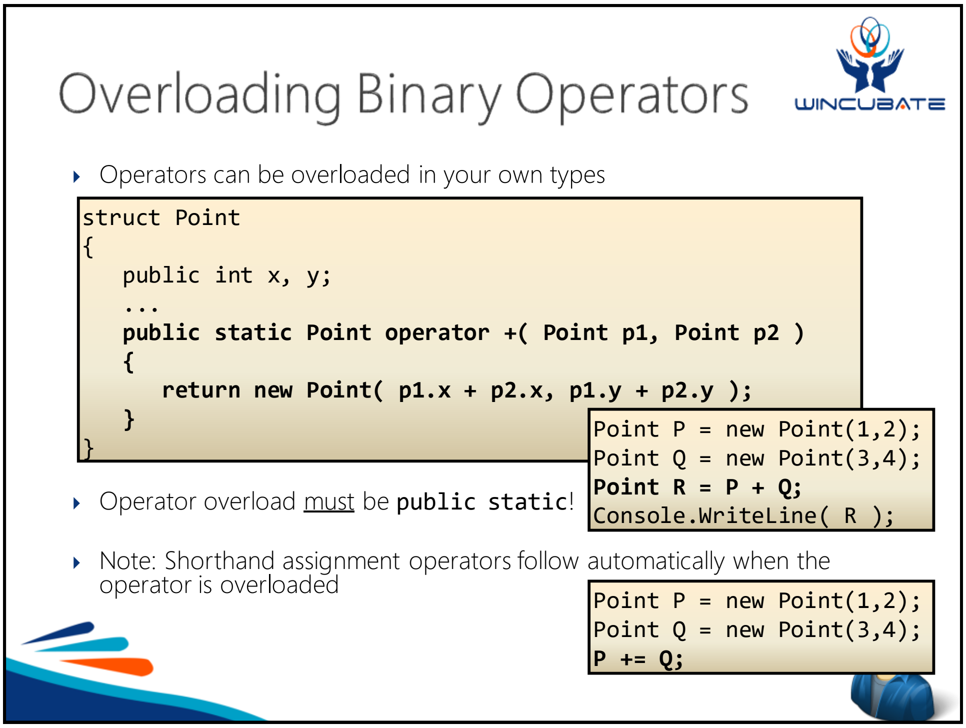


Operator Overloading

Operator	Overloadability	
`+ - ! ~ ++ -- true false`	Unary can be overloaded	✓
`+ - * / % & | ^ << >>`	Binary can be overloaded	✓

`+= -= *= /= %= |= ^=`
`<<= >>=`
	Shorthands follows automatically	⚙

`== != < > <= >=`	Comparisons can be overloaded in pairs	✓
`[ ]`	Create indexers instead	✗
`( )`	Create type conversions instead	✗



Overloading Binary Operators

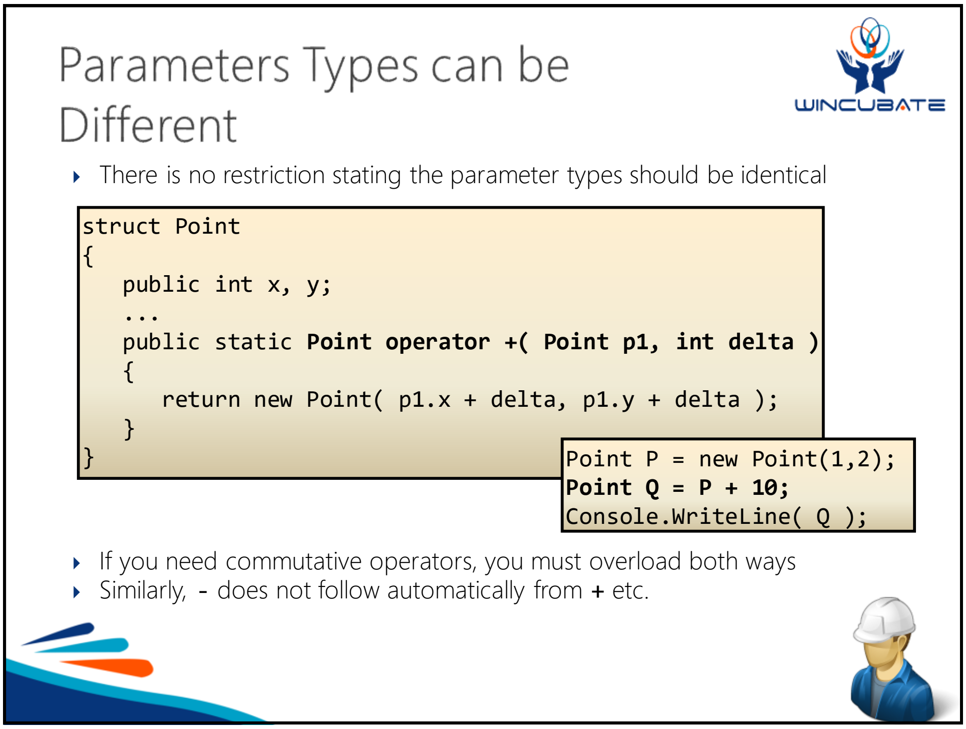
- Operators can be overloaded in your own types

```
struct Point
{
   public int x, y;
   ...
   public static Point operator +( Point p1, Point p2 )
   {
      return new Point( p1.x + p2.x, p1.y + p2.y );
   }
}
```

```
Point P = new Point(1,2);
Point Q = new Point(3,4);
Point R = P + Q;
Console.WriteLine( R );
```

- Operator overload <u>must</u> be `public static`!
- Note: Shorthand assignment operators follow automatically when the operator is overloaded

```
Point P = new Point(1,2);
Point Q = new Point(3,4);
P += Q;
```



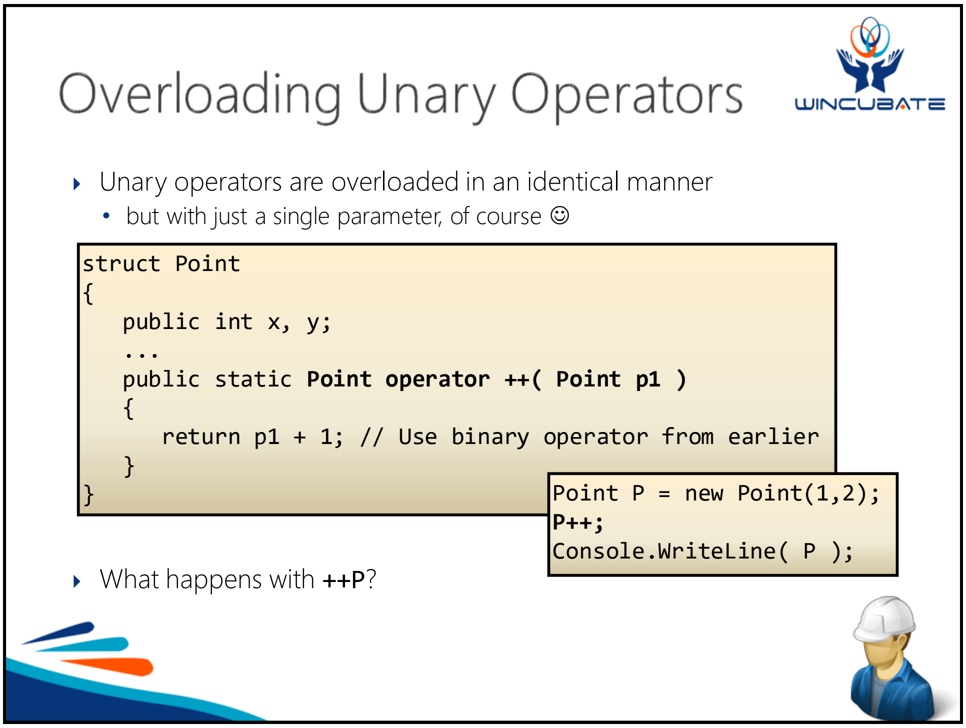
Parameters Types can be Different

- There is no restriction stating the parameter types should be identical

```
struct Point
{
   public int x, y;
   ...
   public static Point operator +( Point p1, int delta )
   {
      return new Point( p1.x + delta, p1.y + delta );
   }
}
```

```
Point P = new Point(1,2);
Point Q = P + 10;
Console.WriteLine( Q );
```

- If you need commutative operators, you must overload both ways
- Similarly, **-** does not follow automatically from **+** etc.



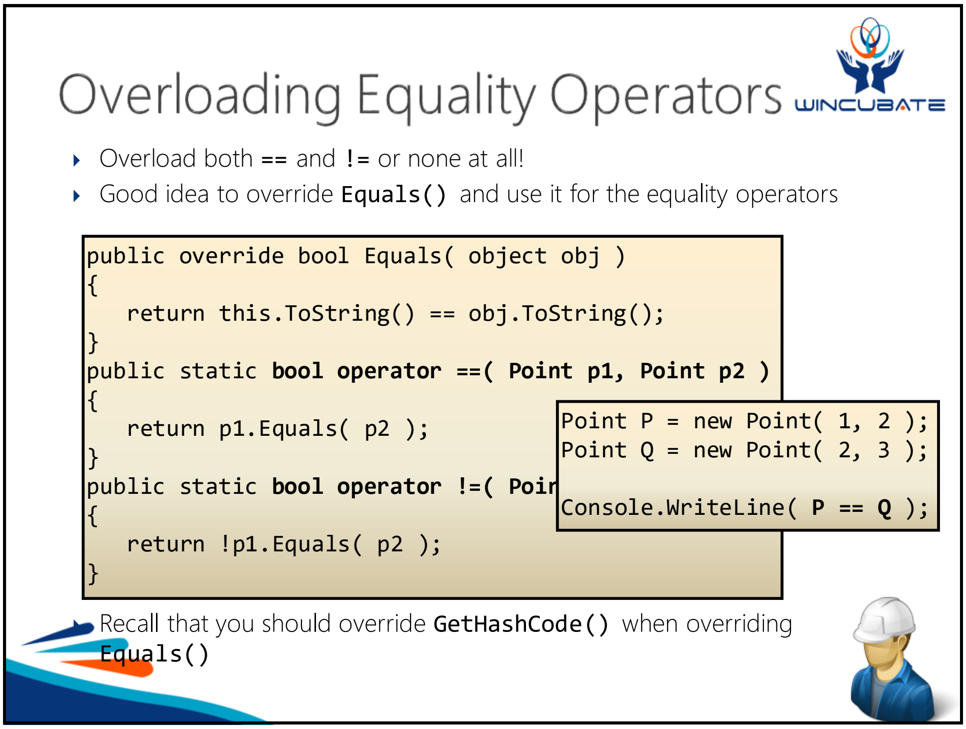
Overloading Unary Operators

- Unary operators are overloaded in an identical manner
  - but with just a single parameter, of course ☺

```
struct Point
{
   public int x, y;
   ...
   public static Point operator ++( Point p1 )
   {
      return p1 + 1; // Use binary operator from earlier
   }
}
```

```
Point P = new Point(1,2);
P++;
Console.WriteLine( P );
```

- What happens with **++P**?



Overloading Equality Operators

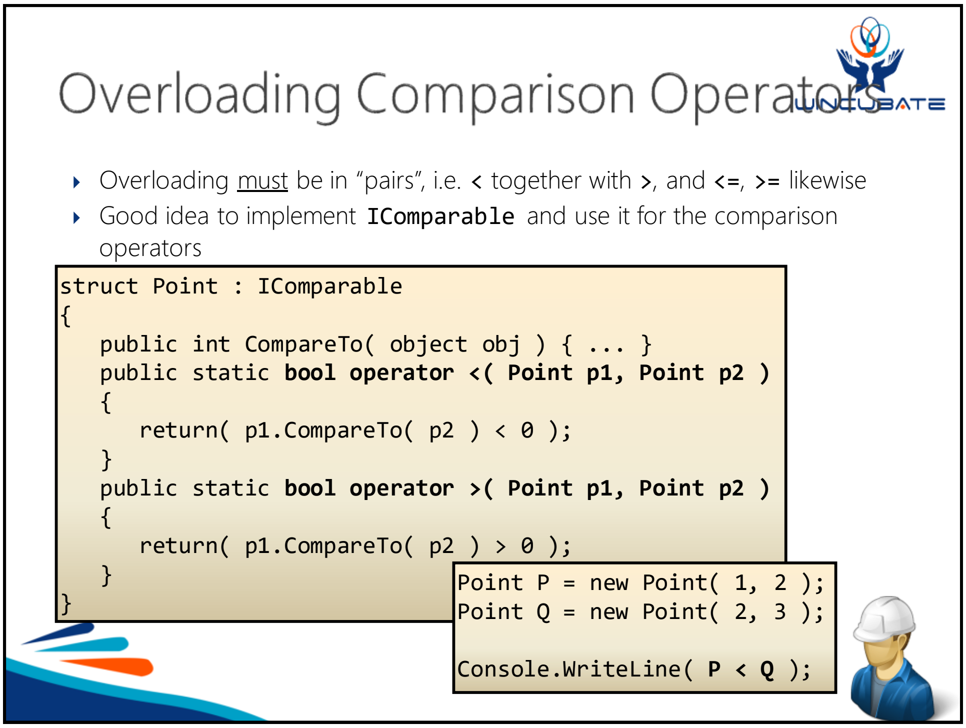
- Overload both `==` and `!=` or none at all!
- Good idea to override `Equals()` and use it for the equality operators

```
public override bool Equals( object obj )
{
   return this.ToString() == obj.ToString();
}
public static bool operator ==( Point p1, Point p2 )
{
   return p1.Equals( p2 );
}
public static bool operator !=( Poir
{
   return !p1.Equals( p2 );
}
```

```
Point P = new Point( 1, 2 );
Point Q = new Point( 2, 3 );

Console.WriteLine( P == Q );
```

- Recall that you should override `GetHashCode()` when overriding `Equals()`



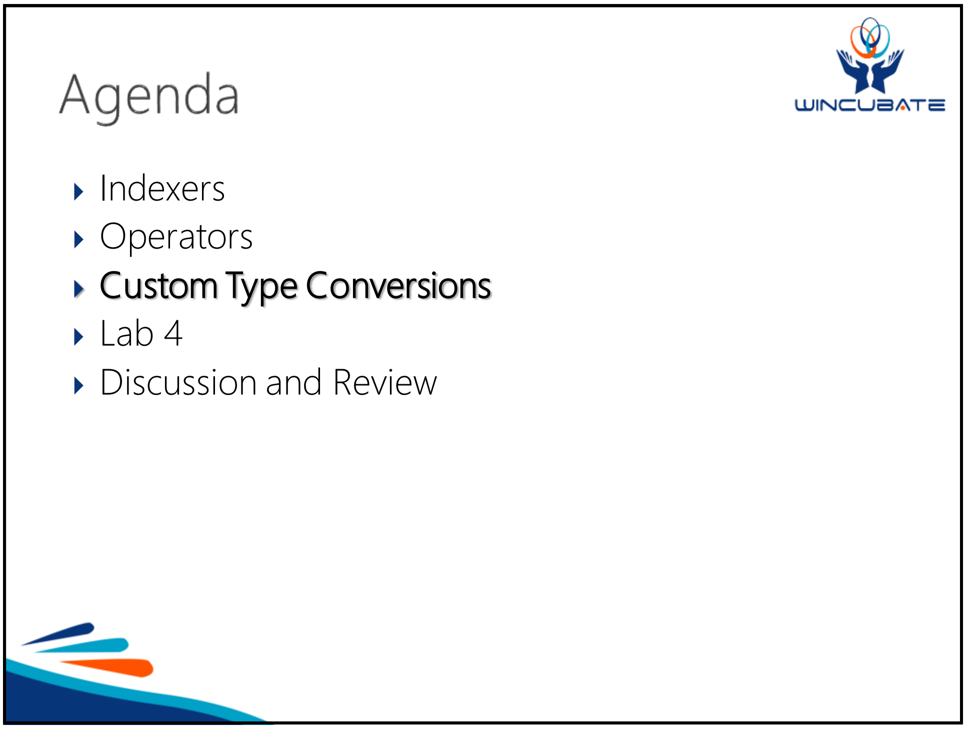
Overloading Comparison Operators

- Overloading <u>must</u> be in "pairs", i.e. `<` together with `>`, and `<=`, `>=` likewise
- Good idea to implement `IComparable` and use it for the comparison operators

```
struct Point : IComparable
{
   public int CompareTo( object obj ) { ... }
   public static bool operator <( Point p1, Point p2 )
   {
      return( p1.CompareTo( p2 ) < 0 );
   }
   public static bool operator >( Point p1, Point p2 )
   {
      return( p1.CompareTo( p2 ) > 0 );
   }
}
```

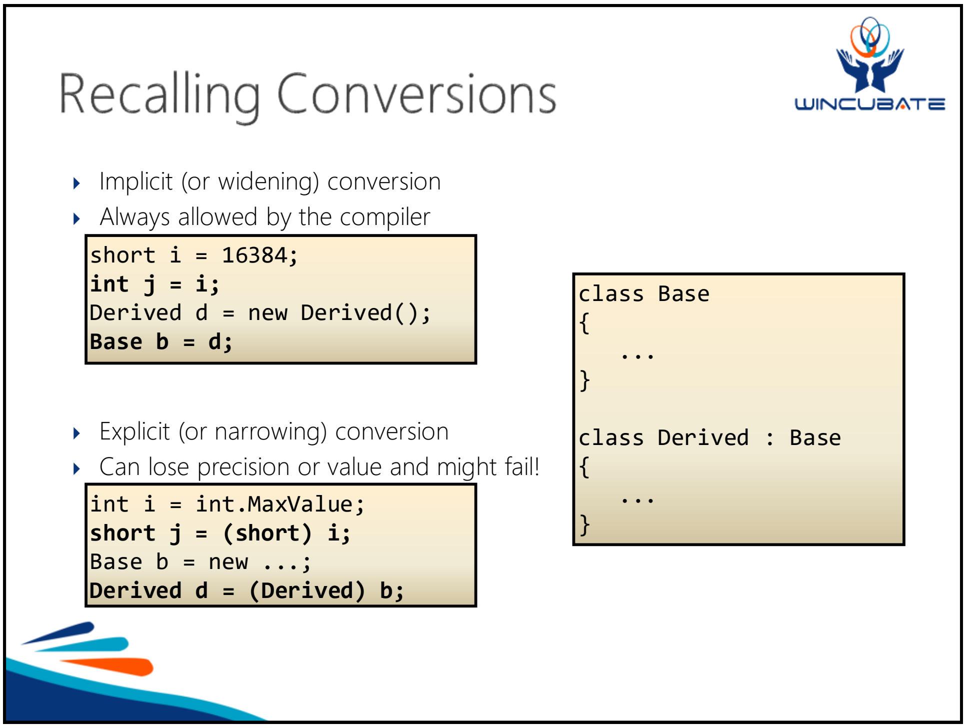
```
Point P = new Point( 1, 2 );
Point Q = new Point( 2, 3 );

Console.WriteLine( P < Q );
```



Agenda

- Indexers
- Operators
- **Custom Type Conversions**
- Lab 4
- Discussion and Review



Recalling Conversions

- Implicit (or widening) conversion
- Always allowed by the compiler

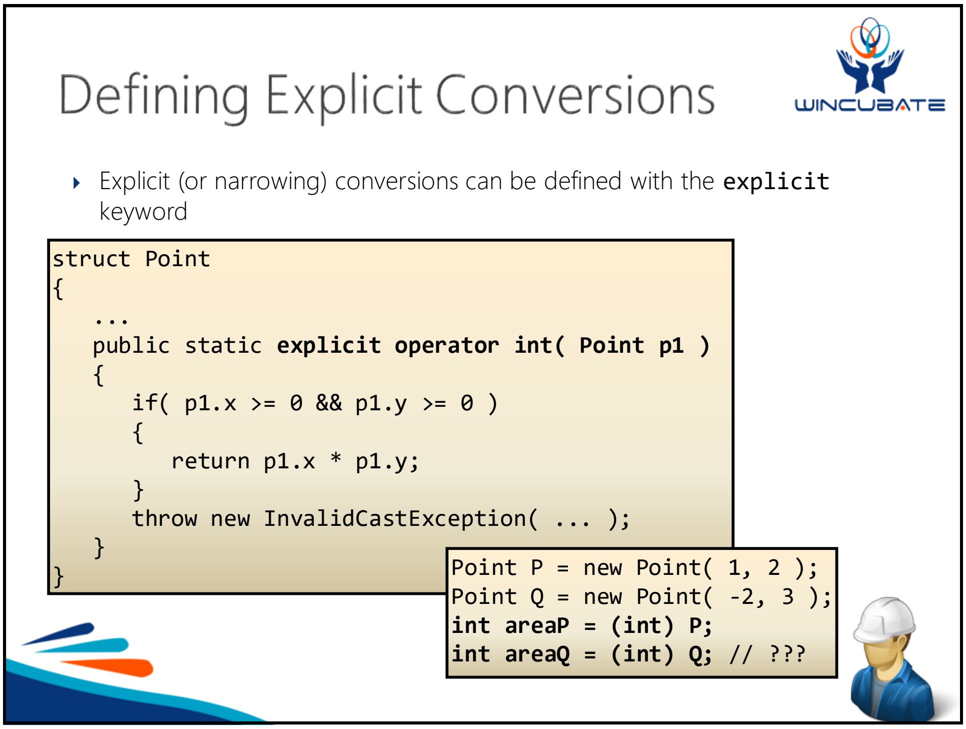
```
short i = 16384;
int j = i;
Derived d = new Derived();
Base b = d;
```

- Explicit (or narrowing) conversion
- Can lose precision or value and might fail!

```
int i = int.MaxValue;
short j = (short) i;
Base b = new ...;
Derived d = (Derived) b;
```

```
class Base
{
   ...
}

class Derived : Base
{
   ...
}
```

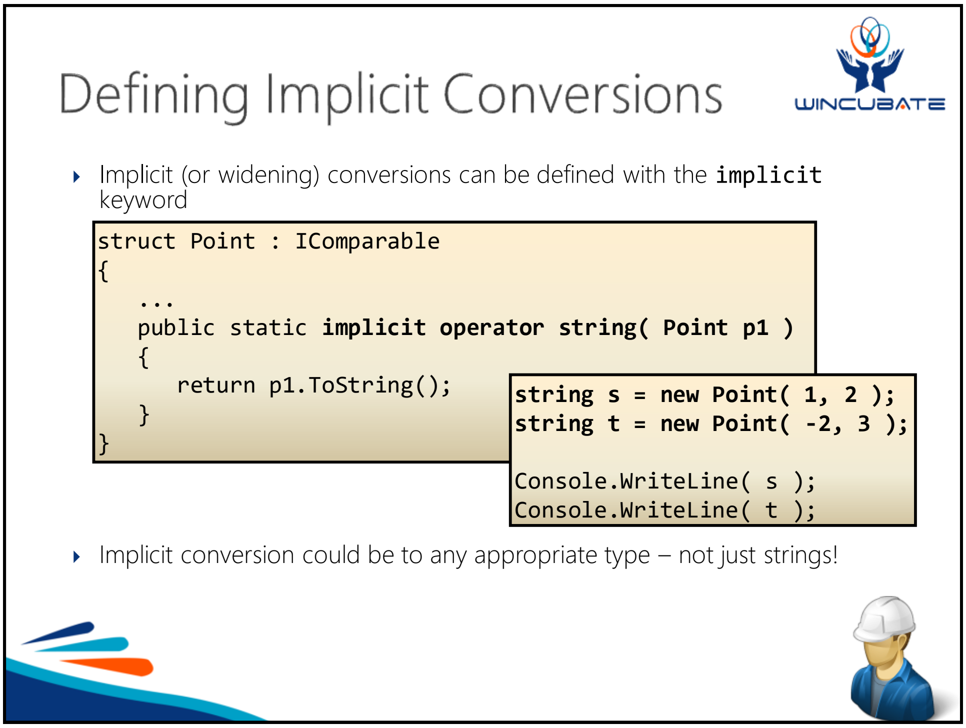


Defining Explicit Conversions

- Explicit (or narrowing) conversions can be defined with the **`explicit`** keyword

```
struct Point
{
   ...
   public static explicit operator int( Point p1 )
   {
      if( p1.x >= 0 && p1.y >= 0 )
      {
         return p1.x * p1.y;
      }
      throw new InvalidCastException( ... );
   }
}
```

```
Point P = new Point( 1, 2 );
Point Q = new Point( -2, 3 );
int areaP = (int) P;
int areaQ = (int) Q; // ???
```



Defining Implicit Conversions

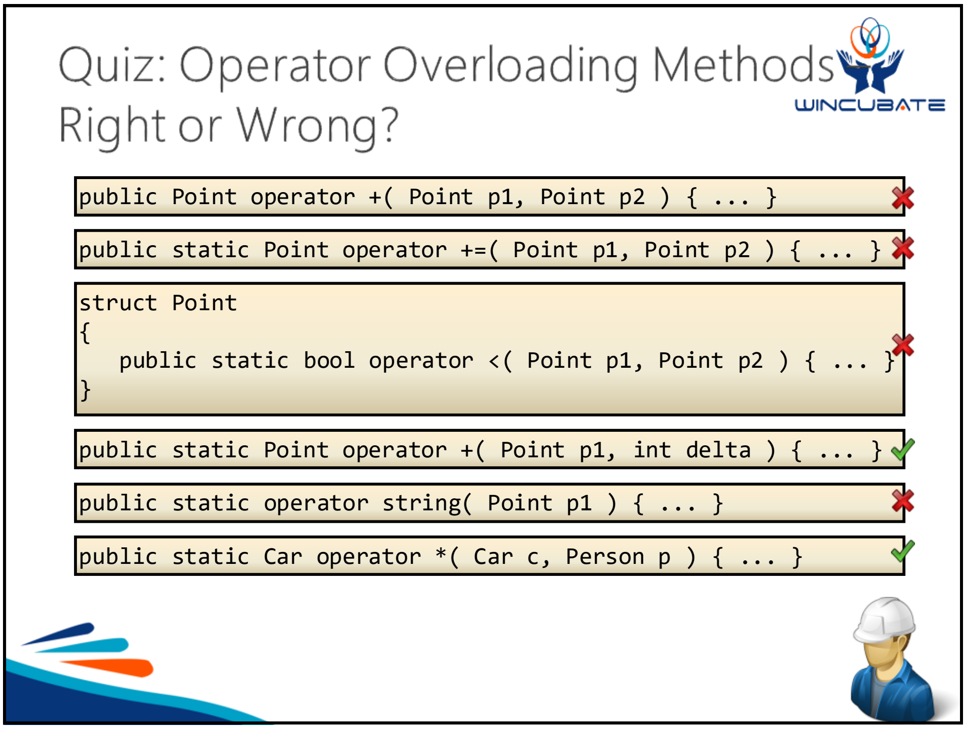
- Implicit (or widening) conversions can be defined with the **`implicit`** keyword

```
struct Point : IComparable
{
   ...
   public static implicit operator string( Point p1 )
   {
      return p1.ToString();
   }
}
```

```
string s = new Point( 1, 2 );
string t = new Point( -2, 3 );

Console.WriteLine( s );
Console.WriteLine( t );
```

- Implicit conversion could be to any appropriate type – not just strings!



Quiz: Operator Overloading Methods Right or Wrong?

```
public Point operator +( Point p1, Point p2 ) { ... } ✗
```

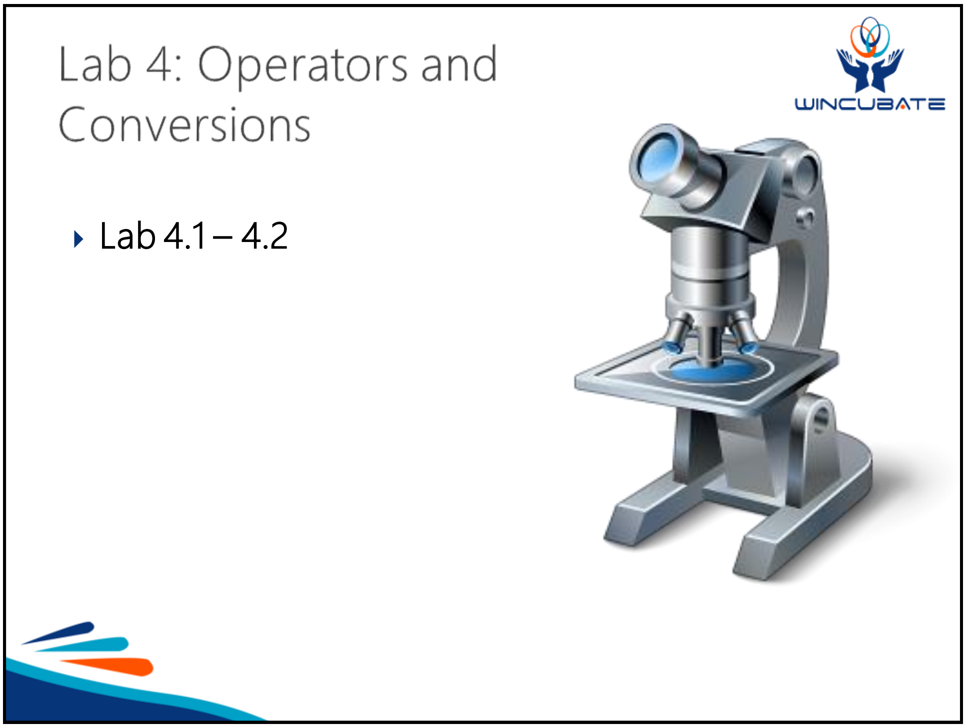
```
public static Point operator +=( Point p1, Point p2 ) { ... } ✗
```

```
struct Point
{
   public static bool operator <( Point p1, Point p2 ) { ... } ✗
}
```

```
public static Point operator +( Point p1, int delta ) { ... } ✓
```

```
public static operator string( Point p1 ) { ... } ✗
```

```
public static Car operator *( Car c, Person p ) { ... } ✓
```

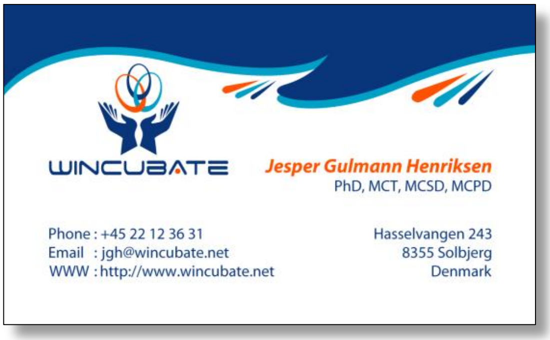


Lab 4: Operators and Conversions

- Lab 4.1 – 4.2

Discussion and Review

- Indexers
- Operators
- Custom Type Conversions



WINCUBATE

Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD

Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net

Hasselvangen 243
8355 Solbjerg
Denmark