



# COMPSCI 280 S1 2011 Enterprise Software Development

Programming Fundamentals  
Arrays & Strings

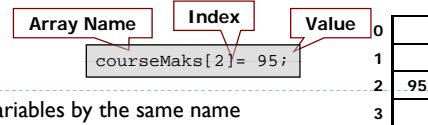


## Agenda & Reading

- ▶ **Agenda:**
  - ▶ Arrays
    - ▶ Arrays of Objects
    - ▶ Multidimensional Arrays
    - ▶ 2D Arrays
    - ▶ Jagged Arrays
  - ▶ Strings
    - ▶ Creating Strings
    - ▶ Strings Manipulation
    - ▶ Strings Comparison
- ▶ **Recommended Reading:**
  - ▶ Microsoft Visual C# 2008 Step by Step
    - ▶ Chapter 10
  - ▶ C# for Java Programmers
    - ▶ Chapter 4
  - ▶ C# Programming Guide
    - ▶ <http://msdn2.microsoft.com/en-us/library/9b9dty7d.aspx>
    - ▶ <http://msdn2.microsoft.com/en-us/library/ms228364.aspx>
- ▶ **Hands-On Lab:**
  - ▶ Lecture07Lab



## Arrays



- ▶ Arrays allow you to refer to a series of variables by the same name
- ▶ An array can be Single-Dimensional, Multidimensional or Jagged.
- ▶ **Array Overview**
  - ▶ Array elements
    - ▶ All of the variables within an array are called elements and are of the same data type
    - ▶ The default value of numeric array elements are set to zero, and reference elements are set to null.
  - ▶ Length
    - ▶ The number of elements in an array
    - ▶ It determines the amount of memory allocated for the array elements. (can't be changed)
  - ▶ Index/Subscript
    - ▶ You can access the individual variables in an array through an index or subscript
    - ▶ Subscript numbering begins at 0 and the last element in an array (length-1)
  - ▶ Array Dimensions
    - ▶ The dimensionality or rank corresponds to the number of subscripts used to identify an individual element. (max=32)
  - ▶ Array types are reference types derived from the abstract base type Array
- ▶ **Declaring Arrays:**
  - ▶ Example:

```
int[] hours;
```

declares an array variable but does not assign an array to it = null



## Creating Arrays

- ▶ To create an array using the New clause
  - ▶ Declare a variable, create the array & initialize the array elements to their default values
    - ▶ Memory space allocated
    - ▶ Array elements are initialize to their default values

```
int[] nb1 = new int[3];
```

size

Size = 3  
Index: 0 to 2

|   |   |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |

- ▶ To create an array and supply initial element values in the New clause
  - ▶ Declare a variable, create the array & initialization

```
int[] numbers2 = new int[3] {1, 2, 3};
```

- ▶ Short cut:

```
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

- ▶ **Note:** It is possible to declare an array variable without initialization, but you must use the new operator when you assign an array to this variable

```
int[] array3; //declare
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK

//array3 = {1, 3, 5, 7, 9}; // Error
```



# Using Arrays

- To access the individual variables in an array through an index

```
Example: nb1[0] = 1;
```

|   |   |
|---|---|
| 0 | 1 |
| 1 | 0 |
| 2 | 0 |

- Information:**

- To determine the number of elements of an array

```
Console.WriteLine(nb1.GetLength(0));
Console.WriteLine(nb1.Length);
```

Parameter:  
The dimension/rank

- To determine the highest subscript value (upper bound) of an array

```
Console.WriteLine(nb1.GetUpperBound(0));
```

- Array processing is easily done in a loop

- A for loop is commonly used,

```
for (int i = 0; i < nb1.Length; i++) {
    numbers1[i] = i;
}
```

```
foreach (int x in nb1)
    Console.WriteLine(x);
```

- You also can use foreach statement

- It runs the statement block for each element in a collection, instead of a specified number of times.
- It uses an element variable that represents a different element of the collection during each repetition of the loop

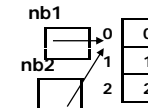


# Using Arrays (con't)

- Assigning Arrays**

- The new array variable holds a copy of the reference held in the variable numbers. So, there is still only one copy of the array

```
int[] nb2;
nb2 = nb1;
```



- Using the Array class**

- Provides methods for creating, manipulating, searching, and sorting arrays
- Use Array.Sort method to sort arrays in ascending order

```
int[] numbersForSorting = {7, 12, 1, 6, 3};
Array.Sort(numbersForSorting);
string[] names = {"Sue", "Kim", "Alan", "Bill"};
Array.Sort(names);
```

- Use Array.Reverse method to reverse the order

```
Array.Reverse(names);
```

- Use Array.Copy method to copy an array

- Parameters: the original, new array and the number of elements

```
int[] copy = new int[3];
Array.Copy(nb1, copy, 3);
```



# Copying Arrays

- Method 1:**

- Copy the elements one at a time in a loop

- Method 2:**

- Use the Array.copy method

- It Copies a range of elements from an Array starting at the specified source index and pastes them to another Array starting at the specified destination index.

```
Array.Copy(srcArray, srcIndex, destArray, destIndex, length);
```

- Method 3:**

- Use the CopyTo method from the original array

- Parameters: the new array and the starting index

- Method 4:**

```
nb1.CopyTo(copy, 0);
```

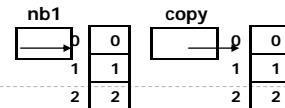
- Use the Clone method from the original array

- An array can be cloned to reproduce an exact copies of the original
- Type casting is needed

```
copy = (int[])nb1.Clone();
```

- Changing the contents of the copied object would not affect the contents of the source object

```
nb1[0] = 10;
```



# Arrays of Objects

- Apart from arrays of value types we can have arrays of reference types

- Example:**

- To create an array of MyPoint elements

- To create an array

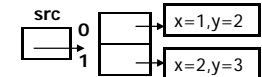
```
MyPoint[] src = new MyPoint[2];
```

- Note: Memory space are allocated for the array
- Each of which is initialized to a null reference

- To create the object elements

- Invoke the new keyword to create the MyPoint object

```
src[0] = new MyPoint(1, 2);
src[1] = new MyPoint(2, 3);
```



- Note:**

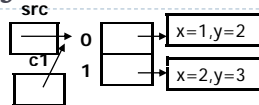
- In an array of reference types not the objects are stored in the array, but references to the objects.
- The objects themselves have to be created and memory space has to be allocated for them separately.



# Working with Arrays of Objects

## Assigning arrays

```
MyPoint[] c1;
c1 = src;
Console.WriteLine(c1 == src);
```



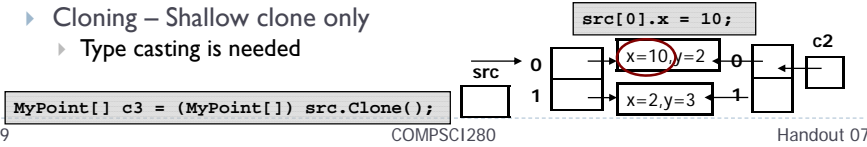
- The variable c1 holds a copy of the reference held in the variable src. There is still only one copy of the array.
- Thus changing the object pointed to by one of the variables will also cause the contents of the other variable to change

## Copying arrays

```
MyPoint[] c2 = new MyPoint[2];
Array.Copy(src, c2, 2);
```

- Use Array.Copy to copy arrays
- However both source and destination elements now refer to the same object (pointing to the same physical object in memory)
- Thus changing the element will also cause the contents of the other variable to change

- Cloning – Shallow clone only
  - Type casting is needed



# Multidimensional Arrays

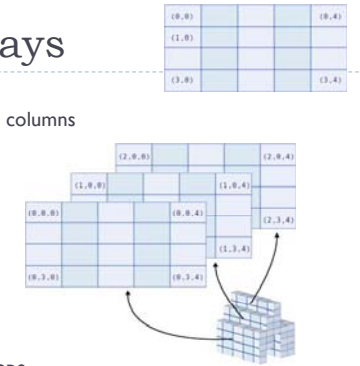
## You can declare arrays with up to 32 dimensions

- A Two-dimensional array is like a table with rows and columns

```
int[,] rectangle = new int[4, 5];
```

- A Three-dimensional array is like a cube,
  - with rows, columns, and pages

```
int[,,] cube = new int[5, 4, 5];
```



## Overview

- The rank/dimension of an array is held in its Rank property
- The lowest subscript value for a dimension is always 0
- The length of each dimension is returned by the GetLength method
  - Note that the argument you pass to GetLength and GetUpperBound (the dimension for which you want the length or the highest subscript) is 0-based.
- The highest subscript value is returned by the GetUpperBound method for that dimension.
- The length property of the array is the total size of an array

```
Console.WriteLine(rectangle.Length);
Console.WriteLine(rectangle.GetLength(0));
Console.WriteLine(rectangle.GetUpperBound(0));
```



# Creating and Using 2D Arrays

## Creating Arrays

- To create an array using the New clause
  - Declare a variable, create the array and initialize with default values

```
double[,] weights = new double[2, 2];
```

- To create an array and supply initial element values in the New clause

```
int[,] nums2 = new int[,] { {1, 2, 3}, {4, 5, 6} };
int[,] nums3 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |

## Accessing elements

```
Console.WriteLine(nums2[rowToGet, colToGet]);
```

- To access elements, we use pretty much the same technique as a 1D array

## Navigating elements

- Nested loops are used to navigate the array elements

```
for (int i = 0; i < nums2.GetLength(0); i++){
  for (int j = 0; j < nums2.GetLength(1); j++){
    Console.Write(nums2[i, j] + " ");
  }
  Console.WriteLine();
}
```

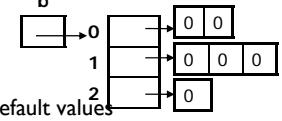


# Jagged Arrays

- An array of which each element is itself an array is called an array of arrays. The elements of a jagged array can be of **different** sizes.

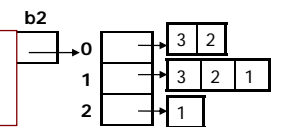
## Creating Jagged Arrays

- To create a Jagged Array
  - Declare a variable, create the array and initialize with default values



```
int[][] b = new int[3]{};
b[0] = new int[2];
b[1] = new int[3];
b[2] = new int[1];
```

single-dimensional array that has three elements, each of which is a single-dimensional array of integers



- To create an array and supply initial element values in the New clause

```
int[][] b2 = new int[][] { new int[] {3,2}, new int[] {3,2,1}, new int[] {1} };
```



# Using Jagged Arrays

## Length & UpperBound

- The jagged array is ONE dimension only.
- Each element of the jagged array is itself an array.
- The rank/dimension of an array is held in its Rank property.
- The highest subscript value is returned by the GetUpperBound(0) method.
- The length property of the jagged array returns the **number of arrays** contained in the jagged array.
- The GetLength(0) method returns the number of elements

## Accessing elements

- To access individual elements like this example, which displays the value of the second element of the first array.

```
Console.WriteLine(b[0][1]);
```

## Navigating elements

- Note: b2[i].GetLength(0) returns the number of the array element at the array index i

Number of arrays

Number of elements of the corresponding array contained in the jagged array

```
for (int i = 0; i < b2.Length; i++){
    for (int j = 0; j < b2[i].GetLength(0); j++){
        Console.Write(b2[i][j] + " ");
        Console.WriteLine();
    }
}
```



# Strings

- A C# string is an array of characters declared using the string keyword.
- string** is an alias for String in the .NET Framework.
- String objects are read-only and immutable, meaning that they cannot be changed once they have been created.

## Creating strings

- Using literal
- Using C# string constructors

```
string greeting = "Hello";
```

```
string repeated = new string('c', 4);
Console.WriteLine(repeated);
```

```
char[] charArray = { 'h', 'e', 'l', 'l', 'o' };
string fromCharArray = new string(charArray);
```

## Working with Strings

- Escape Characters
  - \n, \t ...
- The @ Symbol
  - ignore escape characters and line breaks

```
string hello = "Hello\nWorld!";
```

```
string p1 = "\\My Documents\\My Files\\";
string p2 = @"\\My Documents\\My Files\";
```



# Strings Manipulation

## Length

- Returns the length of the string

```
Console.WriteLine(greeting.Length);
```

## Basic operations:

- Retrieves a character
  - Using its index
- Concatenate strings using the + operator

```
char a = greeting[0];
Console.WriteLine(greeting += a);
```

'H'  
"HelloH"

## IndexOf()

- To search for a string inside another string
- Parameter: The String to seek.
- returns
  - 1 if not found
  - the zero-based index of the first location at which it occurs.

10  
-1

```
string s1 = "Battle of Hastings, 1066";
Console.WriteLine(s1.IndexOf("Hastings"));
Console.WriteLine(s1.IndexOf("1967"));
```

## Replace()

- Replaces all occurrences of a specified char or String in this instance, with another specified Unicode character or String.

```
Console.WriteLine(p1.Replace("Documents", "Pictures"));
```



# Strings Manipulation (con't)

## Trim()

- Removes all occurrences of a set of specified characters from the beginning and end of this instance
- No parameter: remove all white space
- Parameter: the char to remove

```
string str1 = " *|@123***456@|;*";
string delim = " *|@";
string str2 = str1.Trim(delim.ToCharArray());
```

123\*\*\*456

## SubString()

- Retrieves a substring from this instance
- Parameters: starting position and/or the length

```
string s4 = "Visual C# Express";
Console.WriteLine(s4.Substring(7, 2));
```

C#

## Split

- Returns a String array containing the substrings in this instance that are delimited by elements of a specified Char or String array.
- Parameter: delimited element

```
char[] delimiter = new Char[] { ' ', '.' };
string words = "this is";
string[] split = words.Split(delimiter);
```

this  
is

this, " ", " ", is, " "



## Strings Manipulation (con't)

### Concat

- Concatenates one or more instances of string

```
Console.WriteLine(String.Concat(words, " good"));
```

Static method

### Join

- Concatenates a specified separator String between each element of a specified String array, yielding a single concatenated string.
- Parameters: separator and a string array

```
string[] s5 = { "apple", "orange", "grape" };
Console.WriteLine(String.Join(", ", s5));
```

apple, orange, grape  
Static method

### ToLower/ToUpper

- Returns a copy of this String converted to lowercase/uppercase

### ToCharArray

- Copies the characters in this instance to a Unicode character array.



## Strings Comparison

### Equals

- String.Equals(Object)

- Determines whether this instance of String and a specified object, which must also be a String object, have the same value (value equality)
- This method performs an ordinal (case-sensitive and culture-insensitive) comparison.

- String.Equals(String, String)

- Static method
- Determines whether two specified String objects have the same value.

### Use the equality/inequality operator (==, !=)

### Use IsNullOrEmpty to check for a null reference or an empty string

```
str1 = "ABCD";
str2 = "abcd";
Console.WriteLine(String.IsNullOrEmpty(str1));
Console.WriteLine(str1 == str2);
Console.WriteLine(str1.Equals(str2));
Console.WriteLine(String.Equals(str1, str2));
```

Static method

False



## Strings Comparison (con't)

### Using string.CompareTo

```
result = str1.CompareTo(str2);
```

- returns an integer value based on whether one string is less-than (<) or greater-than (>) another.
- Performs a word (case-sensitive and culture-sensitive) comparison using the current culture. (e.g. "a" < "A")

### Using string.Compare

- Static method
- Parameters: two strings objects, and/or bool, StringComparison enumeration
  - bool
    - IgnoreCase
      - Indicates that the string comparison must ignore case
    - StringComparison Enumeration
      - Ordinal
        - Compare numeric values of the corresponding Char objects in each string ('A' < 'a')
      - OrdinalIgnoreCase
        - must ignore case, then perform an ordinal comparison (=if they were converted into uppercase, then compares the Unicode code points)

```
result = String.Compare(str1, str2);
```

+ve



## Examples:

```
str1 = "ABCD";
str2 = "abcd";
```

### Ignore Case

```
result = String.Compare(str1, str2, true);
```

+ve

### Compare by Ordinal

```
result = String.Compare(str1, str2, StringComparison.Ordinal);
```

-ve

```
result = String.CompareOrdinal(str1, str2);
```

### Compare by OrdinalIgnoreCase

```
result = String.Compare(str1, str2, StringComparison.OrdinalIgnoreCase);
```

=0

### Compare with null reference

- Any string, including the empty string (""), compares greater than a null reference; and two null references compare equal to each other.
- Always use the static compare method when comparing null reference. The CompareTo method raises an exception if the string is being tested is null.

```
result = String.Compare(str1, null);
result = String.Compare("", null);
result = String.Compare(null, null);
```

+ve

+ve

=0