Programming in C# I

Properties

Contents:

- Properties instead of setter and getter methods
- get and set accessors
- Using properties
- The contextual keyword value.
- Auto-implemented properties.

Farid Naisan, University Lecturer, Malmö University, farid.naisan@mau.se

Communication between objects of classes

- Object of classes need to use each other in an application to get things done.
- When a class uses an object of another class, it must create an object of the other class (unless it is a static class) using the keyword new.
- The class can then access all the public members declared in the object's class.
- Every object that is created will have its own set of values in the object's fields,and it has its own copy of all the methods declared in their class.
- The fields (instance variables) of a class should be private (or protected in some cases) according to principles of object-orientation.
- The fields, however, need to be accessed by other objects to get new values or provide information about the values they currently hold.
- I languages other than C# and VB, programmers use the so called setter and getter methods to let this communication take place.

Setter and getter method

- A setter method is used to change the value of variable with a new value.
- The instance variable name is a string in the example below.
- The setter method has a **void** return type but has a parameter with a new value for **name** – same type as the **name** (string).
- The getter method has a **string** return type same as the **name**, but no argument, as it only needs to returns a value.

```
public class Product
{
    private string id;  //Id to be set in the ProductMnager
    private string name;  //name of a product != empty
    private double price;  //price, >= 0.0

    MoreCode

    //A setter method example - in C# use the set-property instead
    public void SetName(string name)
    {
        if (string.IsNullOrEmpty(name))
            this.name = name;
    }

    //A getter method example - in C# use the get-property instead
    public string GetName()
    {
        return name;
    }
```


Use properties in C# instead of setter and getter

- In C# the setter and getter methods have been standardized and are known as Properties.
- A property is connected to an instance variable and it should have the same return type as the instance variable.
- It is has accessors, get and set which replaces the getter and setter methods, respectively.
- A get-method is used to return the value saved in the instance variable to the caller, and a set-method is used to save a new value in the instance variable. The new value comes through an identifier called value.
- Note that the word value is a keyword but only in the context of properties. It is not a keyword outside properties.
  - A contextual keyword is used to provide a specific meaning in the code, but it is not a reserved word in C#.

Properties Example

- **Name**, in the code shown below, is a property that gives both read and write access to the instance variable name.
- It is very common to use the variable name with a capital first letter for the property meant for the variable.
  - The Name property connected to name.
  - The identifier value get the same type as the Property's type, e.g. string in the example

```
public class Product
{
    private string name;  //name of product
    private string id;    //unique id, generated by the appl.
    private decimal price;  //cost of the product

    //property connected to instance variable name
    //read and write access
    public string Name
    {
        get
        {
            return name;
        }
        set  //changes value of name
        {
            if (!string.IsNullOrEmpty(value))
                name = value;
        }
    }
```

get { } provides read-access

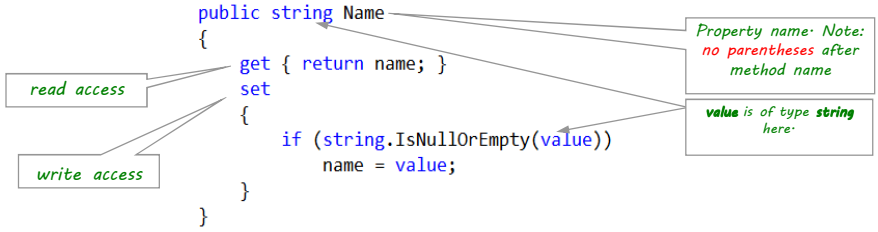
set { } provides write-access

Properties

- There are two types accessors in a Property:
  - get – to read (return) value of a field.
  - set – to change or update value of a field (write).
- It is pretty common to refer to the accessors as get-property and set-property, which we will also use hereafter.
- You can have one or both of these connected to a field. It is common to refer to
- If you only provide a get-property, the field becomes ”read-only”.
- If you have only a set-property, the field becomes ”write-only”.
- The examples in the next slides show both of these forms.

Get and set example – The contextual keyword value

- The keyword `value` gets the same type as the return value for the property. If the return value is an object type, `value` will also be an object of the same type.
- **Name** (with a 'N') is the Property name (could be called GetName as well) and has a return type `string` (because it is connected to the variable **name** (with a small 'n').
- It is a sort of a convention that a Property-name is selected close to the name of the variable it is connected to (Name - name).



Only get- or set property example

```
//Only read access - no set Property
public string ID
{
    get { return id; }
}
//Only write access - no get Property
public string Notes
{
    set { notes = value; }
}
```

value has the type string here.

Calling properties

- Properties are used in the same way as instance fields, i.e. with dot notation, **objName.PropertyName**.
- The compiler knows exactly which of the properties, `get` or `set`, to invoke, depending on the call.
- Rule of thumb: when a property-name is assigned a value, the `set` property is called:

```
dishWasher.Price = 9000.0;
```

- When the value of a property is retrieved, the `get` property is called.

```
double moms = 0.25 * dishWasher.Price;
```

- **Note** that there is no parenthesis after the property name. Writing parentheses, will cause a compiler error (ex .Price()), as the compiler considers the call as a method call. This is also how you can differentiate a method call from a property call. Properties have no parentheses – methods do.

Initialization of objects using properties

- You can create an instance of a class without calling a parameterized constructor as shown in the code example.
- Name, Price and Count are properties defined in the class Product.
- Note that there is a comma not a semicolon after each initialization.

```
static void Main()
{
    Product product = new Product
    {
        Name = "Dishwasher 5XL",
        Price = 499.9, //Euro
        Count = 1
    };
}
```

Auto-properties

- C# allows auto-implemented properties as shown on lines 34-36 in the example code.
- The compiler creates a private variable and connects to the Property. The variable will be anonymous and not seeable in the code.

```
void Test()
{
    // Intialize a new object.
    Customer customer = new Customer(5000, "Northwind", 90100);

    // Modify a property.
    customer.TotalPurchases += 499.99;
}
```

```
29  // This class is mutable. Its data can be modified from
30  // outside the class.
31  class Customer
32  {
33      // Auto-implemented properties for trivial get and set
34      public double TotalPurchases { get; set; }
35      public string Name { get; set; }
36      public int CustomerId { get; set; }
37
38      // Constructor
39      public Customer(double purchases, string name, int id)
40      {
41          TotalPurchases = purchases;
42          Name = name;
43          CustomerId = id;
44      }
45      // Methods and additional methods, events, etc.
46  }
```


Using auto-properties vs standard properties

- A good practice is to validate the "value" before accepting its contents, as in the examples here.
- Auto-implemented properties are good to use when you do not have logics in the set and get methods.
- Auto-properties are like using public instance variables (fields) but not the same thing.
  - Auto properties can be re-written as standard properties without the need for the client code using it to be affected.
  - Never use public fields! It is against OOP laws.

```
class Product
{
    private string name;
    private double price;
    private int count;

    public string Name
    {
        get
        {
            return name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
                name = value;
        }
    }
    public double Price
    {
        get
        {
            return price;
        }
        set
        {
            if (value > 0.0)
                price = value;
        }
    }
}
```

Summary

- Properties are used to provide other objects access to values stored in the object's private fields.
  - The get property sends out the value saved in instance variable (read access).
  - The set property gives other objects access to change the value of a field variable (write access).
  - Both actions can take place under controlled conditions.
- A method call has always a parentheses with or without parameters:
  - name = GetName(); //method call
  - name = Name; //Property call
- A Property call has not parentheses and no parameters:
  - name =Name;
- A practice is to validate value in the set-accessor of a property.

C# Programming

One-Dimensional Arrays

Contents:

- One-dimensional arrays
- Array operations
- Arrays as return value and method parameter
- foreach statement
- Copying, initializing and manipulating arrays

Farid Naisan, Senior University Lecturer, Malmö University, farid.naisan@mau.se

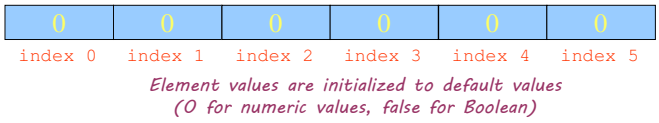
Introduction to Arrays

- Primitive variables are designed to hold only one value at a time.
- An array is a variable that can hold a collection of values of the same type which are indexed. Each value is an element of the array and has an index.
- In C# and most other languages indexing begins with 0. The last index is the length of the array – 1.

- An array's length is the number of elements the array can store.
- An array is a list of elements with the same data type. It can store any type of data, int, double, bool, objects, but the same type for all elements.
- A position in the array does not have to have a value, it can be empty or null (objects).

Creating Arrays

- An array is an object and every object in C# needs to be created using the keyword new.



Element values are initialized to default values (0 for numeric values, false for Boolean)

- The variable holding the array is actually a reference to a space (address) in the memory, and is called a reference variable.
- A reference variable is first declared and then assigned the address of the object by creating the object with the keyword new.

```
//Declare a reference to an array that will hold integers.
int[] numbers;  //Array declared but not created
```

- The next step is to create the array object, `numbers` with a capacity to store 6 elements indexed 0 to 5.

```
numbers = new int[6];  //The array is created and ready to store values
```


Two-dimensional arrays

- The array in the previous slide represents a one-dimensional array. It is like a list of values.
- An array can have two dimensions like a table. An element in a two dimensional array are is referred to by a horizontal index and a vertical index. It is easier to think of row and column.
- The value -12 in the table has the position [1,1] which points to second row and second column in the table.
- Arrays can have more than two dimensions which is not very usual but possible.
- In this presentation, we concentrate wholly on one-dimensional arrays and leave the two-dimensional variant to a separate discussion.

Index	0	1	2	3
0	33	56	42	70
1	11	-12	0	0
2	22	0	0	0
3	33	-34	-50	77
4	439	889	79	29

Creating Arrays

- Syntax:

```
dataType[] variableName; //declare a variable
variableName = new dataType[numOfElements]; //create the array
```

- The declaration and creation of an array object can be done on one line as in the code snippet below:
- The variable primes can store 100 int-values.
- The variable prices can store 5000 double values.
- The variable persons can store 150 objects (instances) of the class Person (the class not shown here).

```
class ArrayTest
{
    public void Test()
    {
        int[] primes = new int[100]; //index 0 to 99
        double[] prices = new double[5000]; //index 0 to 4999
        bool[] emptySlots = new bool[200];
        string[] names = new string[100];

        //Array with element of the class Person
        Person[] persons = new Person[150];
    }
}
```


Array size (capacity)

- The array size must be a non-negative number.
- It may be a literal value (100) or a constant (arraySize).

```
const int arraySize = 100;
int[] numbers = new int[arraySize];
```

- Arrays can also be created with size that is a variable.

```
void CreateArray ( int size )
{
    double[] priceList = new double[size];
}
```

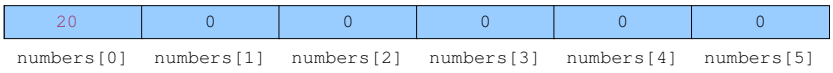
- Once created, an array size is fixed and cannot be changed; that is why arrays are said to be static to differ from dynamic arrays which do not have a fixed size.
- Memory will be allocated for the size of the array no matter how many positions are actually used.

Accessing the Elements of an Array

- An array is accessed by the reference name (variable name) and an index (subscript) within straight brackets that identifies which element in the array to access.

```
arrayVariable [index]
```

- The index must be >= 0 and less the length of the array. If not, it will cause a run-time error which may cause the program to crash.




```
numbers[0] = 20; //pronounced "numbers sub zero"
```


Inputting and Outputting Array Elements

- Array elements can be treated as any other variable.
- The difference is use of index. In fact, an array is a collection of variables having the same variable name.
- Each variable is accessed by the same name and a subscript.
- It is possible to create an array with size that is a variable. For example, the size can be input from the caller method or given by the user (at run time).

```
const int size = 450;
double[] priceList = new double[size];

public double  GetElementAt(int index)
{
    return priceList[index];
}
```

```
int numOfTests;     // The number of test scores
int[] testScores;       // Array of test scores
Console.Write("How many numbers do you have? ");
numOfTests = Convert.ToInt32(Console.ReadLine());

//Create an array for the test scores
testScores = new int[numOfTests];
```

Off-by-One Errors

- A common beginner mistake is to be off-by-one when working with arrays – because of the "<=" operator!

```
// This code has an off-by-one error.
int[] numbers = new int[100];

for (int i = 1; i <= 100; i++)      //Error
                                    //To correct:  change to i < 100
    numbers[i] = 99;
```

- Here, the equal sign allows the loop to continue on to index 100, where 99 is the last index in the array.
  - The index goes out of range (out of 0 to 99) and the code will throw an exception which is a run-time error.

Initialization of the of elements of an array

- When an array is created, the elements of the array are initialized by the compiler using the default value of the element-type.
- Arrays of string are initialed to null until they are assigned a value in the code. Using an element that is null will cause a run-time error.
- Generally, when an array has elements that are objects (classes), every element must be created using the keyword new.

Type of array	Elements are initialized to	Comments
Numerical types like int [ ], double [ ], etc.	0 (or 0.0)	Value types cannot be set to null
bool [ ]	false	Value type, can only be true or false
string [ ]	null	string is an object
All other objects	null	objects cannot be set to 0

Initialization of a whole array

- An array can be initialized holding some constant values.

```
int[] days = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

- The numbers in the list are stored in the array in the following order:

```
days[0] is assigned 31,
days[1] is assigned 28,
days[2] is assigned 31,
days[3] is assigned 30,
etc.
```

- Array initialization in this manner is very convenient when constant values like number days in months of the year (or names of months, days of the week) or other factors are to be saved in an array.
- It is also an efficient way of testing a code with initial values.

Initialization specifying dimension

- The dimensions can specified, (although there is no reason to do it) much in the same as one-dimensional arrays.

```
string[,] names = new string [3, 2]{ { "one", "two" }, { "three", "four" }, { "five", "six" } };
```

- All of the following declarations provide an array with the same dimension, i.e. an array with 12 elements.

```
int[] days1 = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

int[] days2 = new int[12] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

int[] days3 = new int[] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

- If the dimension is given as with days2, the compiler will check so the size match the count of elements specified in the initialization list.
- The first alternative, days1, can only be used at the declaration time, whereas the other two alternatives, the array can be initialized after declaration;

```
int[] days2;
days2 = new int[12] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Array Length

- Arrays are objects implying that they have methods and properties (properties comes in a later module).
- They have a public property named **Length** that returns the number of elements in the array. This is useful in case you create an array whose size is a variable that you do not save.

```
double[] temperatures = new double[25];
```

- The length of this array is 25 and can be retrieved at any time in the code by using the Lenth proper of the array.

```
int size = temperatures.Length; //size = 25
```

- The length of an array is useful to check the upper-bound of the array (Length-1) and use it to make sure the array is not indexed with a value that is out of range.

Working with Array Elements

- Using and manipulating of data in an array element is the same as any other variable with the only difference that the value is accessed by an index.

```
totalToPay = numbers[5] * payRate;
```

- Pre and post increment works the same:

```
nt[] score = { 7, 8, 9, 10, 11 };

int value = ++score[2] + 5;  //Pre-increment operation, i.e. the value of score[2] is
                             //incremented by 1 before it is used

int value = score[4]++;      // Post-increment operation, the current value of score[4]is
                             // used first and then score [4] is increased by 1.
```

- Assume index = 1 (score[1] is 8).

```
sum = score[index++]; //sum = 8, index is then 2, index is first used then incremented
```

Array Size

- The Length property can be used in a loop to set the number of iterations.

Index subscripts start at 0 and ends at one less than the array length

```
for (int i = 0; i < temperatures.Length; i++)
{
    Console.WriteLine ( "Temperature {0}: {1}", i, temperatures[i] );
}
```

Important:
If you use <= than you must adjust the condition: i <= temperatures.Length-1;
Run-time error: i <= temperatures.Length; //compiler cannot help here.

Adding, changing and removing an element

- When an array is created with a number of elements, then the memory is reserved and no element can be deleted (as long as the array lives).
- To remove an element is to mark the element as empty, or not used in some way.
  - Array of strings – initialize the elements to string.Empty or even null.
  - Array of numbers – can differ depending of what values can be used to mark an element as empty (array of positive number can be initialized to -1, as an example.
- Let's run an exam in which we use an array of double values to store prices of something.
- The same pattern can be used for other value types, or objects.

Example – add, remove and change

- The size of the array priceList is determine by the caller method when calling the method CreateArray.
- To make sure that the size is not 0 or negative, we validate the argument and signal back to the caller if the array could be successfully created.
- To begin with, the array is initialized by the compiler to 0s. Let's change to -1 because a price can be 0 but should not be -1.
- The method Reset (line 28) initiates all position to -1, and use the method on line 23
- This way, we can detect if a position in the array is empty and can have a new value.

```
9  class ProductManager
10 {
11
12     private double[] priceList;
13
14     //Create array with a given size
15     public bool CreateArray(int size)
16     {
17         bool ok = false;
18         if (size > 0)
19         {
20             priceList = new double[size];
21             ok = true;
22         }
23         ResetArray();
24         return ok;
25     }
26     //initiate all element to -1
27     //-1 says that elements are empty
28     private void ResetArray()
29     {
30         for (int i = 0; i < priceList.Length; i++)
31             priceList[i] = -1.0;
32     }
33
```


Add a new element in the priceList array

- To a new array we should find the first vacant position. A vacant position where the value is -1 (as we planned in the previous slide).
- We write a method for this job (Line 51) and use it in the Add method on line 40.
- The **FindAnEmptyPosition** method loops through the array and when it encounters an element having the value -1, it stops the iteration and saves the index for returning to the caller.
- If it does not find an empty position, then the array must be full. The index remains to be -1 which will be returned.
- The Add method know if the return value on line 42 is not -1, an empty position is found it set ok = true; otherwise it signals a false value to the caller.

```
34 //Add a new element if the array is not full
35 public bool Add(double newValue)
36 {
37     bool ok = false;
38     //Find the first vacant position
39     //check so he array is not full
40     int index = FindAnEmptyPosition();
41
42     if (index != -1)
43     {
44         priceList[index] = newValue;
45         ok = true;
46     }
47
48     return ok;
49 }
50
51 private int FindAnEmptyPosition()
52 {
53     int index = -1; //not found
54     for (int i = 0; i < priceList.Length; i++)
55     {
56         if (priceList[i] == -1) //if empty
57         {
58             index = i; //empty pos found
59             break;   //cancel iterations
60         }
61     }
62     return index; //will -1 if not found
63 }
```

Change or remove an existing element

- To change the value at a position or remove (mark as empty) an element, we must have the index of the element to process the job.
- To avoid a possible program crash, we must always ensure so the index is not out of bounds.
- The method CheckIndex on Line 79 validates a given index.
- The method on line 66 is responsible for performing the change. We must have the index of the element and the new value.
- For removing an element, we only need the index. Removing means to mark the position as empty (or vacant, or unused), Line 88.

```
65  //change an existing value at position = index  with a new value
66  private bool ChangeElementAt(int index, double newValue)
67  {
68      bool ok = false;
69      //validate index
70      if (CheckIndex(index))
71      {
72          priceList[index] = newValue; //the old value is overwritten
73          ok = true;
74      }
75
76      return ok;
77  }
78
79  private bool CheckIndex(int index)
80  {
81      bool ok = false;
82      if ((index >= 0) && (index < priceList.Length))
83          ok = true;
84
85      return ok;
86  }
87
88  public bool RemoveElementAt(int index)
89  {
90      bool ok = false;
91      //validate index
92      if (CheckIndex(index))
93      {
94          priceList[index] = -1; //mark the position as empty again
95          ok = true;
96      }
97
98      return ok;
99  }
```


Resizing an array

- An array reference can be assigned to another array of the same type.

```
// Create an array referenced by the numbers variable.
int[] numbers = new int[10];
// Reassign numbers to a new array
numbers = new int[5];
```

- The last assignment in above code may look like a "resizing" but it is not!
  - The variable **numbers** releases its 10-objects and instead holds the address of a new array.
- The first array (with 10 elements) no longer has a reference to it, and therefore will be garbage collected.
- Resizing must be done manually by creating a new array and copying the values element for element.

Re-assigning Array References

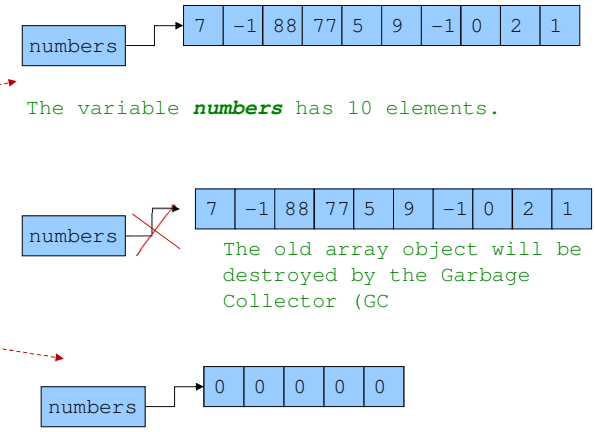
- The array is originally declared to hold 10 elements of int type.

```
int[] numbers = new int[10];
```

- Re-assigning it (within the same scope to another variable) is like re-creating the array reusing the variable name..

```
numbers = new int[5];
```

- A new array but with same name is created.
- The reference variable is re-used but the old array object is lost.

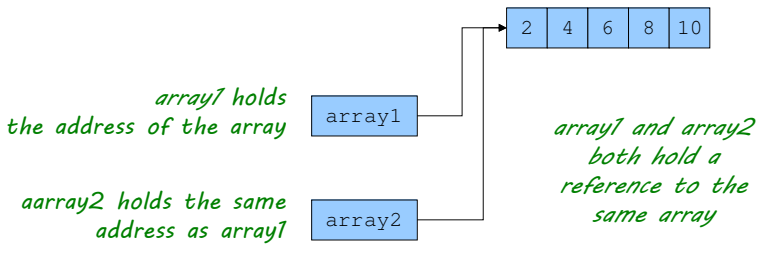


Copying Arrays

- To assign an array variable another variable, we are letting two variables hold the same address in memory – this is not the way to copy arrays.

```
int[] array1 = { 2, 4, 6, 8, 10 };
int[] array2 = array1; // This does not copy array elements.
```

- Arrays must be copied element for element as in the next slide.



Copying Arrays

- You cannot copy an array by merely assigning one reference variable to another (array2 = array1).
- You need to copy the individual elements of one array to another.

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = new int[5];
for (int i = 0; i < firstArray.Length; i++)
    secondArray[i] = firstArray[i];
```

- This code copies each element of **firstArray** to the corresponding element of **secondArray**.

The foreach Loop

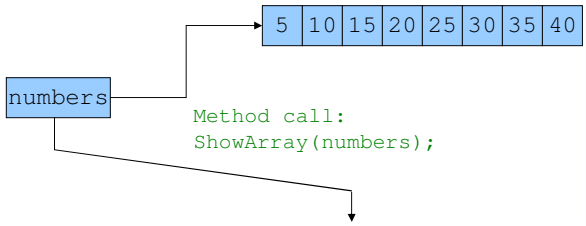
- C# provides an alternative notation for browsing through the array elements, called the **foreach** loop.
- **foreach** is most suitable with collections of objects (ArrayList, List, etc) but it can be used with arrays too.

```
public void Test()
{
    double[] temperatures = { 51.2, 17.5, 18.7, 10.9, -8.0 };

    double sum = 0.0;
    foreach (double temperatue in temperatures)
        sum += temperatue; //get the value directly

    double average = sum / temperatures.Length;
}
```

- The **foreach** loop does not work with the indexes of elements as with other loops. It loops through the array in a sequential fashion and fetches every element directly.
- It cannot be used to access an element at a certain index..
- For ordinary arrays, a **for-statement** or a **while** are better choices, as **foreach** has more overhead.

Passing Array to a method

- When a single element of an array is passed to a method, it is handled like any other single variable.
- More often you will want to write methods to process array data by passing the entire array, not just one element at a time (although this is risky).
- Arrays are objects. Objects are always passed by reference, implying that any changes in the array in the callee method (method that is called) will also affect the array in the caller method.




```
public void ShowArray (int[] array)
{
    for (int i = 0; i < array.Length; i++)
        Console.Write ( "{0} ", array[i] );
}
```


Arrays as return-type

- A method can return an array. The return type of the method must be declared as an array of the right type.
- The **GetArray** method is a public method that returns an array of double values.
- As with other methods returning a value, you will need to use a local variable that matches the return type.
- In the examples here, both methods have local variables (`double[]` and `string[]`) that are arrays and which are then returned to the caller methods.
- Also note that the local arrays are created with a size (by initialization in the first example).

```
public  double[] GetArray ( )
{
    double[] array = { 1.2, 2.3, 4.5, 6.7, 8.9 };
    return array;
}
```

```
class ArrayTest
{
    int[] primes = new int[100]; //index 0 to 99

    public string[] GetPrimeValueStrings()
    {
        int size = primes.Length;
        string[] primeStrings = new string[size];

        for (int i = 0; i < size; i++)
        {
            //format values with 2 decimal positions
            primeStrings[i] = primes[i].ToString("0.00");
        }
        return primeStrings;
    }
}
```

Arrays of Objects

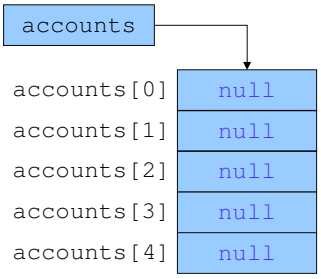
- Elements of an array can be of any type, int, double, string or objects, BankAccount, Customer, etc.

```
BankAccount[] accounts = new BankAccount[5];
```

- The accounts array is an array of references to BankAccount objects.
- Elements of object-types are initiated to null.
- To store an element in the array, the element must be created:

```
accounts[0] = new BankAccount();
```

- If you would like to create all accounts (usually not needed), you can proceed as in this example.




```
void CreateArray (int size)
{
    for (int i = 0; i < accounts.Length; i++)
        accounts[i] = new BankAccount ( );
}
```


Command-Line Arguments using array

- A C# program can receive arguments from the operating system's command-line.
- Looking back at the Main method, it has a header that looks like this:

```
static void Main (string[] args)
```

- The Main method receives a string array as a parameter.
- The values in the **args** array can be used in the main-method as any ordinary one-dimensional array.

```
string firstString = args[0];
```

- The command-line arguments are used in rare cases when the application is started with some parameters.

Summary

- Arrays are a part of the basics of a programming language.
- Arrays represent a list of variables with the same name.
- To access an element in a an array we need to use an index.
- The index is an integer starting from zero for the first element.
- The index to the last element is the number of elements minus 1.
- An array in C# is an object that has a property called Length which returns the number of elements in the array.
- As with other objects, an array must be created using the keyword new. The number of elements are to be given.
- The size of an array remains constant regardless of the number of position that we actually use, and it is therefore an array is said to be static (not meaning array is defined with the keyword static).

Programming in C#

Two-dimensional arrays

Agenda:

- Two dimensional arrays
- Initialization
- Rectangular and jagged arrays
- Multi-dimensional arrays

Farid Naisan, University Lecturer, Malmö University, farid.naisan@mah.se

Two-dimensional Arrays

- A two-dimensional array is similar at matrix or a table consisting of rows and columns.
- When creating the array, the number of rows and columns are given.

```
double [,] numbers = new double[4,5];
```

- The above statement creates a rectangular array with 5 rows and 4 columns.
- To access an element in the array, two subscripts (indexes) are used. The indexes are separated by a comma.
  - `numbers [3, 2] = -50;`
- Apart from that a two-dimensional array uses two indexes, working with it is much similar to a one-dimensional array.
  - Add one more index when creating, initializing the array and accessing an element.
  - Add one more dimension when looping through array.

Index	0	1	2	3
0	33	56	42	70
1	11	-12	0	0
2	22	0	0	0
3	33	-34	-50	77
4	439	889	79	29

Initializing a Two-Dimensional Array

- Initializing a two-dimensional array requires enclosing each row's initialization list in its own set of braces

```
int[,] numbers ={ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

- C# automatically creates the array and fills its elements with the initialization values.
- The above initialization declares an array with 3 rows and 3 columns and creates the following table:

```
1 2 3
4 5 6
7 8 9
```

- Elements of an array are initialized by the compiler to their default values (0, 0.0 for numeric types, false for bool and null for strings and other objects).

Initialization specifying dimension

- The dimensions can specified, (although there is no reason to do it) much in the same as one-dimensional arrays.

```
string[,] names = new string [3, 2]{ { "one", "two" }, { "three", "four" }, { "five", "six" } };
```

- All of the following declarations provide a two-dimensional array with the same size, i.e. 3 rows and 2 columns, and a total count of 6 elements.

```
string[,] names1 =  { { "one", "two" }, { "three", "four" }, { "five", "six" } };
string[,] names2 = new string[3, 2] { { "one", "two" }, { "three", "four" }, { "five", "six" } };
string[,] names3 = new string [, ] { { "one", "two" }, { "three", "four" }, { "five", "six" } };
```

- If the dimensions are given as for `names2`, the compiler will check so the dimensions match the count of elements specified in the initialization list.
- The first alternative, days1, can only be used at the declaration time, whereas with the other two alternatives, the array can be initialized after declaration;

```
string[,] names2;
names2 = new string[3, 2]{ { "one", "two" }, { "three", "four" }, { "five", "six" } };
```

The Length property

- All C# arrays are objects and they have methods and properties.
- One useful property is **Length** which provides the total number of elements in an array. For a two-dimensional array, it return rows * columns as in the example.
- Arrays have also a method, `GetLength(dim)` that returns the number of elements in a dimension (dim) of an array. For two-dimensional arrays, the number of rows and columns can be detected using this method.

```
public void Test()
{
    int[,] numbers = { { 1,  2,  3,  4},
                       { 5,  6,  7,  8},
                       { 9, 10, 11, 12} };

    int totNumOfElements = numbers.Length; //12
    int numOfRows = numbers.GetLength(0);  //3
    int numOfCols = numbers.GetLength(1);  //4

    for (int row = 0; row < numOfRows; row++)
    {
        for (int col = 0; col < numOfCols; col++)
            Console.Write(numbers[row, col] + " ");
    }
    Console.WriteLine();
}
```


Note on two dimensional arrays

- Passing a single or two-dimensional array as an argument to a method is the same as one-dimensional arrays. The method must accept a two-dimensional array as a parameter.
- In the same manner, a two-dimensional array can be used as return value of a method.
- Processing and manipulating of two-dimensional arrays are performed in the same way as with one-dimensional array, keeping in mind working with two dimensions instead of one.
- Arrays declared with the syntax [,] are rectangular which means that all rows have the same number of columns.
- It is also possible to have arrays that have different number of columns. These are known as Jagged arrays.

Jagged Arrays

- When the rows of a two-dimensional array are of different lengths, the array is known as a jagged array; In other words, the length of each array index can differ.
- A jagged array is created as an "array of arrays". The syntax is different compared to rectangular array.
- To create a jagged array, two [ ][ ] are used as in the example.
- In the example code, the size of the first dimension (rows) is given. Each row will then be a new array and needs to be created with a size (as one-dimensional arrays).

```
public void SaveScores()
{
    int teams = 15;

    //array with 15 teams and different
    //num of matches
    int[][] scores = new int[teams][];
    scores[0] = new int[10]; //1st row 10 columns
    scores[1] = new int[7];  //2nd row 7 columns
    scores[2] = new int[11]; //3rd row 11 columns
    //continue.

    scores[0][0] = 3; //Elem at first row first col
    scores[1][6] = 4; //Elem at second row 7th col

}
```


Jagged arrays

- A jagged array is an array where each item in the array is another array.
- It can be created in several ways.
- The size of the first dimension is specified (3 in the first example)
- The array can be created with initial values as in the examples jaggedArray2 and jaggedArray3.

```
int[][] jaggedArray = new int[3][];
jaggedArray[0] = new int[2];
jaggedArray[0][0] = 45;
jaggedArray[0][1] = 77;

jaggedArray[1] = new int[4];
jaggedArray[2] = new int[2];
```

```
int[][] jaggedArray3 =
{
   new int[] { 3, 51, 7, },
   new int[] { 11, 0, 2, 4, 6, 4},
   new int[] { 1, 6,0 },
   new int[] { 15, 10, 2, 42, 6, 45,
   67, 78 }
};
```

```
int[][] jaggedArray2 = new int[][]
{
   new int[] { 3, 51, 7, },
   new int[] { 11, 0, 2, 4, 6, 4},
   new int[] { 1, 6,0 },
   new int[] { 15, 10, 2, 42, 6, 45,
   67, 78 }
};
```

Multi-dimensional arrays

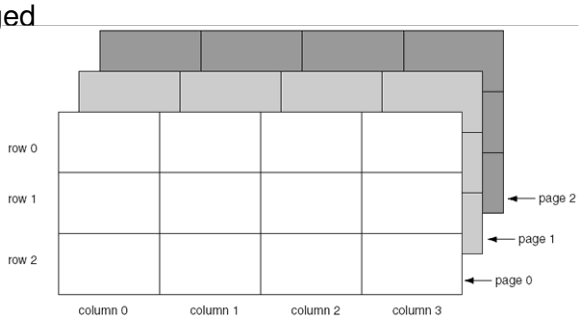
- C# does not limit the number of dimensions that an array may have.
- More than three dimensions is hard to visualize, but may be useful in some programming problems.

```
double[,,] my3DSpace = new double[15, 12, 3];
```

- A multi-dimensional array using a jagged array:

```
double[][,] multiD = new double[5][,];
```

- Multi-dimensional arrays are not used much, as you can use a one-dimension array of objects.
- Good to know about it but no need to spend much time on it.



Jagged arrays vs. multi-dimensional arrays

- A jagged array is said to be an array of arrays (jagged arrays).
- Jagged arrays are faster than multi-dimensional arrays and can be used more effectively.
- Multidimensional arrays have nicer syntax.

```
public void SaveElementAt(int[][] array, int i, int j, int
value)
{
    array[i][j] = value;
}
public  void SaveElementAt(int[,] array, int i, int j, int
value)
{
    array[i, j] = value;

    //create the array with this size
    guestList = new string[maxNumOfGuests];
}
```

Dynamic arrays

- Dynamic arrays utilizes the memory in a much more effective way besides allowing you not to set a fixed size for an array at compile time.
  - The array can grow or shrink at run time depending on the need.
- The size of a dynamic is not set at compile time.
- Every time an element is added to the array, the size is increased by one element and memory is allocated accordingly.
- Every time an element is removed, memory is deallocated and the size is decreased.
- Memory allocation takes place at run time dynamically and therefore the arrays are called dynamic arrays, also known as collections.
- C# takes care of the memory allocation.
- Dynamic arrays are called collections. There are numerous ready-to use collections in the .NET class libraries.
- This topic is not a part of this presentation and will be introduced in another lesson.

Array of objects

- A two dimensional and a multi-dimensional array have elements of the same type, either int, double, or other types.
- There are times that a two or three dimensional array must be used, but when no such requirement prevails, it is a lot more effective and practical to use a one-dimensional array of a class. Each dimension of the array can be an instance variable (field) in the class.
- A product has a name and a price. Using arrays, we need to have two one-dimensional arrays, one for names (string) and one for prices (double).
- If we instead create a class Product and then use an array of Product, it is a much better solution:

```
private Product[] productLisst = new Product[50];
```

```
class Product
{
    private string name;
    private double price;

    // other code
}
```

Summary

- Arrays are important to understand and use as they are among the basic constructions of most programming languages. They are also used in algorithm studies.
- Arrays are objects in C# and must be created as with other objects.
- An array can be one dimensional, two-dimensional or multi-dimensional.
- A two-dimensional array with the same number of columns in all rows is a rectangular array that is like a matrix or a table.
- C# allows to create an array in which each row can contain another array. Such arrays are known as jagged arrays.
- Jagged arrays can be used when two-dimensional arrays have different count of columns.
- Remember, whenever possible use a one-dimensional array of objects as a replacement for two and multi-dimensional arrays, but you MUST have a good understanding and knowledge of two-dimensional arrays as a necessary item in your programming bag-pack.