
ORIANT
C# Collections

Objectives

- Structure
- Class
- Array
  - Array Notation in C#
  - Array Rank
  - Accessing Array Elements
  - Checking Array Bounds
  - Creating Array Instances
  - Initializing Array Elements
  - Single dimensional Array
  - Initializing Multidimensional Array Elements
  - Creating a Computed Size Array
  - Copying Array Variables
  - Array Class Properties
  - Array Class Methods
  - Returning Arrays from Methods
  - Passing Arrays as Parameters
  - 'params' keyword and Param Array
  - Jagged Array
  - Entering Items Into and Displaying Items From Jagged Array
  - Code using Array
- Collections
  - System.Collections
  - Collections – Types
  - Generic Collections
  - Non-Generic Collections
  - Choose a collection
    - Dictionary<TKey,TValue>
    - List<T>
    - Queue<T>
    - Stack<T>
    - LinkedList<T>
    - ObservableCollection<T>
    - SortedList<TKey,TValue>
    - HashSet<T>
    - SortedSet<T>

TIME FOR CASE STUDY

Case Study-2

- As a portal **admin** of Shopon, I want to add more data associated to product like AvailableStatus and Image location with existing data.

Thought

- To add more data, we can add more member data or pass more parameter using ref.
  Let’s code it.

Code

```
class ShoponMain
{

    int pid;
    string productName;
    double price;
    string availableStatus;
    string imageUrl;
    1 reference
    public void Main()
    {
        //1. Read Product Details
        ReadProduct();
        //2. Display Product Details
        DisplayProduct();
    }
```

```
private void DisplayProduct()
{
    Console.WriteLine($"Pid :{pid} \tProduct Name :{productName} " +
                      $"\tPrice :{price}" +
                      $"\tAvailable Status: {availableStatus}" +
                      $"\tImage Url: {imageUrl}");
}
```

```
    private void ReadProduct()
    {
        Console.WriteLine("Enter product id:");
        pid = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Enter product name:");
        productName = Console.ReadLine();
        Console.WriteLine("Enter price:");
        price = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Enter available status:");
        availableStatus = Console.ReadLine();
        Console.WriteLine("Enter image location:");
        imageUrl = Console.ReadLine();
    }
}
```

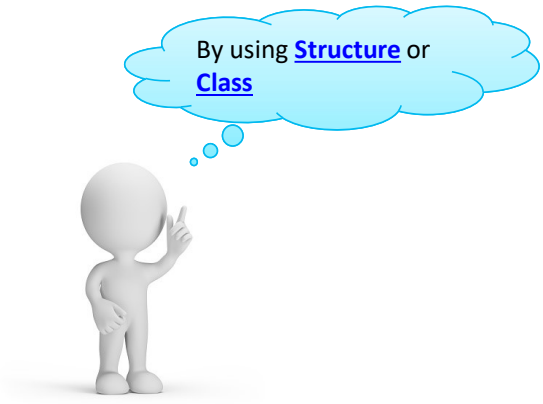
How far this is right way of implementing?

Alternative Solution

Alternative Solution

Any Suggestions?

- Wrapping or encapsulating up the information



Structure

- As we know, structure is value type.

```
struct Product
{
    2 references
    public int Pid { get; set; }
    2 references
    public string ProductName { get; set; }
    2 references
    public double Price { get; set; }
    2 references
    public string AvailableStatus { get; set; }
    2 references
    public string ImageUrl { get; set; }
}
```

```
public void Main()
{
    Product product = new Product();
    //1. Read Product Details
    ReadProduct(product);
    //2. Display Product Details
    DisplayProduct(product);
}
```

```
private void ReadProduct(Product product)
{
    Console.WriteLine("Enter product id:");
    product.Pid = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter product name:");
    product.ProductName = Console.ReadLine();
    Console.WriteLine("Enter price:");
    product.Price = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter available status:");
    product.AvailableStatus = Console.ReadLine();
    Console.WriteLine("Enter image location:");
    product.ImageUrl = Console.ReadLine();
}
```

```
private void DisplayProduct(Product product)
{
    Console.WriteLine($"Pid :{product.Pid} " +
                      $"\tProduct Name :{product.ProductName} " +
                      $"\tPrice :{product.Price}" +
                      $"\tAvailable Status: {product.AvailableStatus}" +
                      $"\tImage Url: {product.ImageUrl}");
}
```


Structure - Output

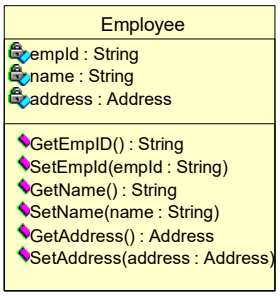
Output


```
Enter product id:
1001
Enter product name:
Note S
Enter price:
23000
Enter available status:
Y
Enter image location:
images/notes.jpg
Pid :0  Product Name :  Price :0       Available Status:       Image Url:
```

NOTE: Structure is value type. The value will be lost between method calls.

Class

- User defined reference type
  - Encapsulates all the data and operations pertaining to an entity
  - Provides a Single representation to all the attributes defining the entity
  - Passing single representations is easier



- Data types as collections
  - A struct in C encapsulates only data. Used as a data structure to store different types of data
  - An array is used to store different elements of the same type

Code

```
class Product
{
    2 references
    public int Pid { get; set; }
    2 references
    public string ProductName { get; set; }
    2 references
    public double Price { get; set; }
    2 references
    public string AvailableStatus { get; set; }
    2 references
    public string ImageUrl { get; set; }
}


public void Main()
{
    Product product = new Product();
    //1. Read Product Details
    ReadProduct(product);
    //2. Display Product Details
    DisplayProduct(product);
}
```

```
private void ReadProduct(Product product)
{
    Console.WriteLine("Enter product id:");
    product.Pid = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Enter product name:");
    product.ProductName = Console.ReadLine();
    Console.WriteLine("Enter price:");
    product.Price = Convert.ToDouble(Console.ReadLine());
    Console.WriteLine("Enter available status:");
    product.AvailableStatus = Console.ReadLine();
    Console.WriteLine("Enter image location:");
    product.ImageUrl = Console.ReadLine();
}

private void DisplayProduct(Product product)
{
    Console.WriteLine($"Pid :{product.Pid} " +
                      $"\tProduct Name :{product.ProductName} " +
                      $"\tPrice :{product.Price}" +
                      $"\tAvailable Status: {product.AvailableStatus}" +
                      $"\tImage Url: {product.ImageUrl}");
}
```


Output

```
Enter product id:
1001
Enter product name:
Note S
Enter price:
23000
Enter available status:
Y
Enter image location:
images/notes.jpg
Pid :1001       Product Name :Note S    Price :23000    Available Status: Y     Image Url: images/notes.jpg
```

CASE STUDY (CONTINUED)

Case Study(continued)

- As a portal **admin** of Shopon, I want to add more products

Thought

- To store multiple products
  - Array
  - Collections

Array

- An array is a data structure that contains several items of the same type.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- When declaring an array, the square brackets ([]) must come after the type, not the identifier.
- Array types are reference types derived from the abstract base type Array. Since this type implements IEnumerable and IEnumerable<T>, we can use foreach iteration on all arrays in C#.

Integer index 0 (zero)

Integer index 4 (four)

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/

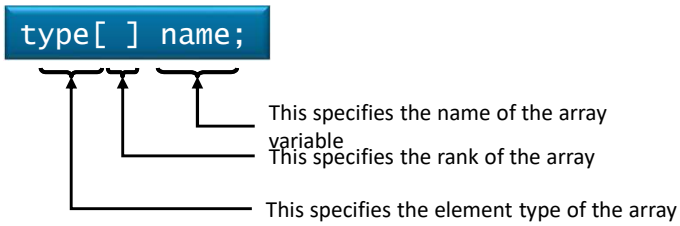
Notes:

An array has the following properties:

- An array can be Single-Dimensional, Multidimensional or Jagged.
- The default value of numeric array elements are set to zero, and reference elements are set to null.
- A jagged array is an array of arrays, and therefore its elements are reference types and are initialized to null.
- Array elements can be of any type, including an array type.

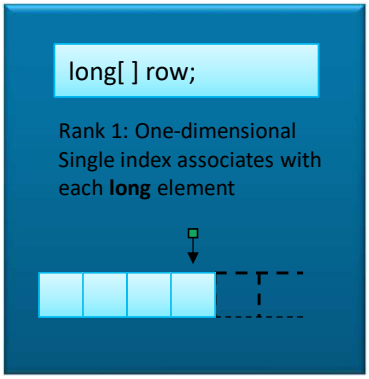
Array Notation in C#

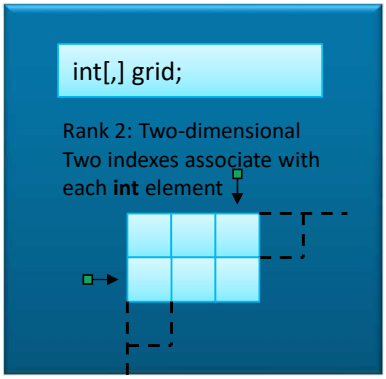
- We can declare an array variable by specifying:
  - The element type of the array
  - The rank of the array
  - The name of the variable



Array Rank

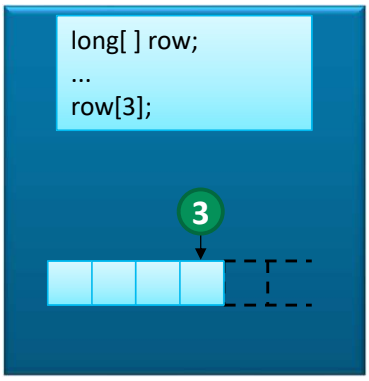
- Rank is also known as the array dimension
- The number of indexes associated with each element

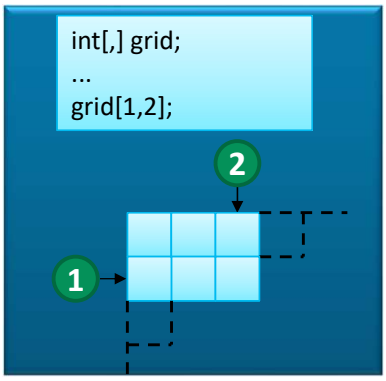




Accessing Array Elements

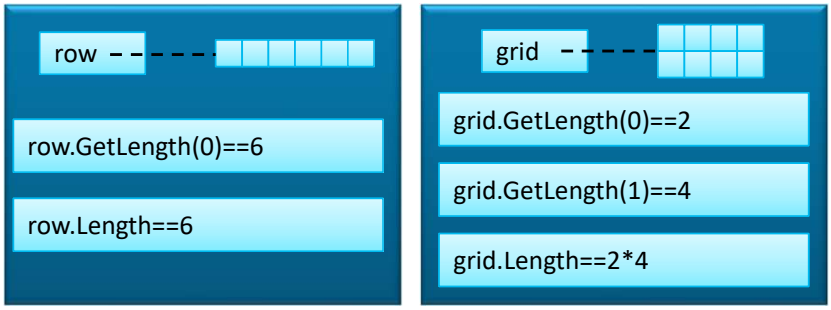
- Supply an integer index for each rank
  - Indexes are zero-based





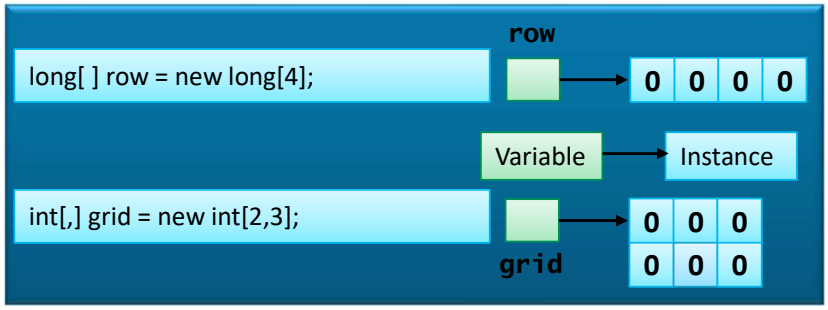
Checking Array Bounds

- All array access attempts are bounds checked
  - A bad index throws an IndexOutOfRangeException
  - Use the **Length** property and the **GetLength** method



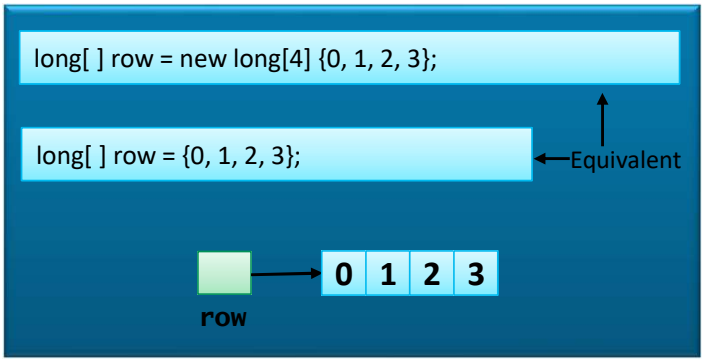
Creating Array Instances

- Declaring an array variable does not create an array!
  - You must use **new** to explicitly create the array instance
  - Array elements have an implicit default value of zero



Initializing Array Elements

- The elements of an array can be explicitly initialized
  - You can use a convenient shorthand

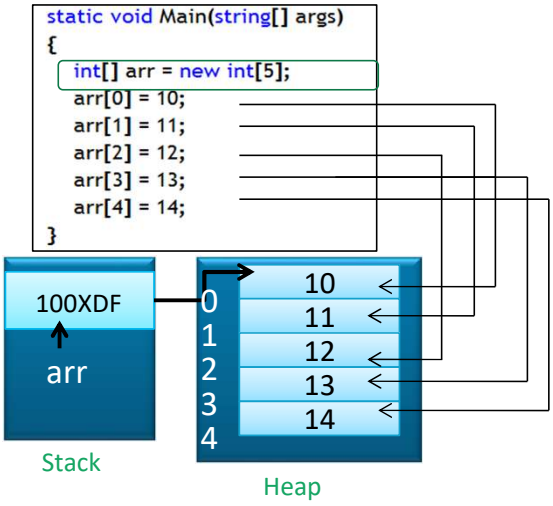


Single dimensional Array

- Array containing a single row of values

Example:

```
int[] numbers = new int[5] {1, 2, 3, 4, 5};
string[] names = new string[3] {"Matt",
"Joanne", "Robert"};
```



Different type of single-dimensional array declaration and initialization:

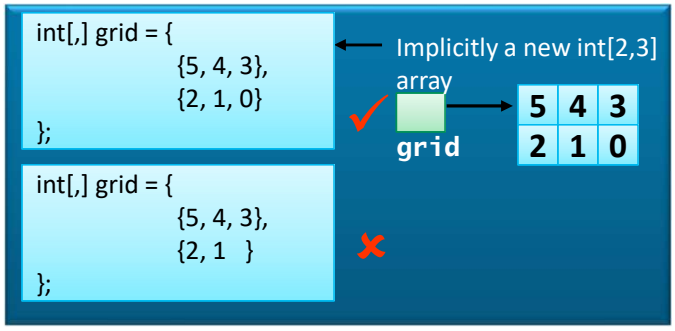
1. Example: int[] numbers = new int[4];
   numbers[0]=1; numbers[1]=2; numbers[2]=3; numbers[3]=4;

2. Example: int[] numbers = new int[] {1,2,3,4};

3. Example: int[] numbers = {1,2,3,4};

Initializing Multidimensional Array Elements

- Array containing multiple rows with same number of items for each row.
- We can also initialize multidimensional array elements
  - All elements must be specified



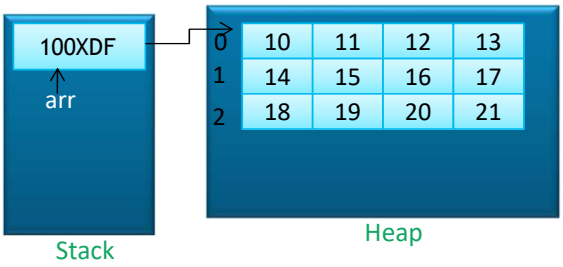
Example:

```
int[,] numbers = new int[3, 4];

numbers[0, 0] = 10; numbers[0, 1] = 11;
numbers[0, 2] = 12; numbers[0, 3] = 13;

numbers[1, 0] = 14; numbers[1, 1] = 15;
numbers[1, 2] = 16; numbers[1, 3] = 17;

numbers[2, 0] = 18; numbers[2, 1] = 19;
numbers[2, 2] = 20; numbers[2, 3] = 21;
```



Different type of multi-dimensional array declaration and initialization:

1. Example: string[,] persondetails= new string[2,3];
   persondetails[0,0]="Mahesh"; persondetails[0,1]="TTG";
persondetails[1,0]="Pramod"; persondetails[1,1]="CC";

2. Example: string[,] persondetails = new string[,] { { "Mahesh", "TTG" },{"Pramod","CC"} };

3. string[,] persondetails = { { "Mahesh", "TTG" }, { "Pramod", "CC" } };

Creating a Computed Size Array

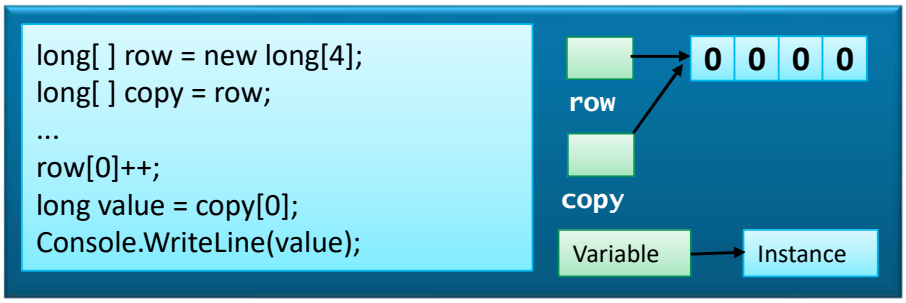
- The array size does not need to be a compile-time constant
  - Any valid integer expression will work
  - Accessing elements is equally fast in all cases
    - Array size specified by compile-time integer constant:
    - Array size specified by run-time integer value:

```
long[ ] row = new long[4];
```

```
string s = Console.ReadLine();
int size = int.Parse(s);
long[ ] row = new long[size];
```

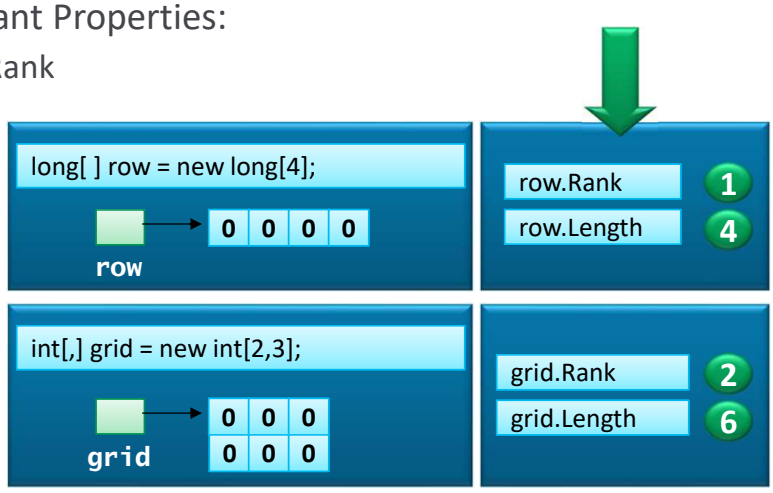

Copying Array Variables

- Copying an array variable copies the array variable only
  - It does not copy the array instance
  - Two array variables can refer to the same array instance



Array Class Properties

- Some Important Properties:
  - Length and Rank



Array Class Methods

Commonly used methods

- **Clear**: Sets a range of elements to zero or **null**
- **Clone:** Creates a copy of the array
- **Copy**: It is a static method. Copies the Array with data to another array.
- **CopyTo**: Copies the Data only to another Array.
- **CreateInstance**: It is a static method Allows to create an Instance of a new Array. This function is used to create Arrays using System.Array class. (To Create Arrays Dynamically).
- **GetLength**: Returns the length of a given dimension
- **IndexOf**: Returns the index of the first occurrence of a value
- **Sort**: Sorts the elements in an array of rank 1

Returning Arrays from Methods

- We can declare methods to return arrays

```
class Example {
   static void Main( ) {
      int[ ] array = CreateArray(42);
      ...
   }
   static int[ ] CreateArray(int size) {
      int[ ] created = new int[size];
      return created;
   }
}
```


Passing Arrays as Parameters

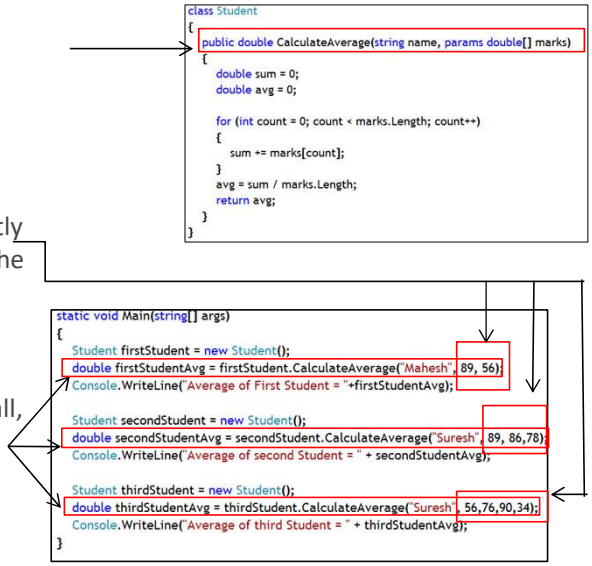
- An array parameter is a copy of the array variable
  - Not a copy of the array instance

```
class Example2 {
   static void Main( ) {
      int[ ] arg = {10, 9, 8, 7};
      Method(arg);
      System.Console.WriteLine(arg[0]);
   }
   static void Method(int[ ] parameter) {
      parameter[0]++;
   }
}
```

This method will modify the original array instance created in Main

‘params’ keyword and Param Array

- ‘params’ keyword is used with an array declaration provided the array is part of the method argument
- This enables the array to accept values directly that is passed to the method directly when the method is called
- While passing parameter during a method call, if the caller is not sure about how many parameters to pass then the method should possess a param array to accept unknown number of arguments

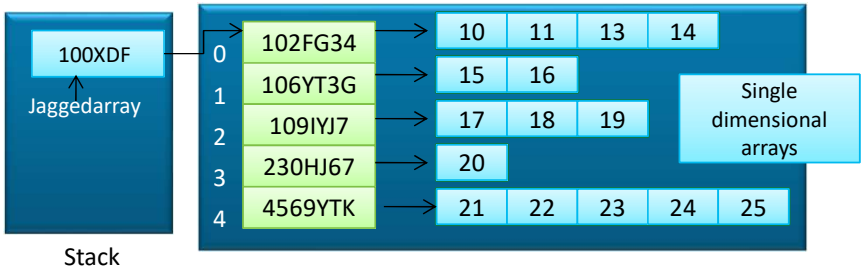


Notes:

1. Only an array declared as part of the method argument list can be declared with ‘params’ keyword
2. The array is known as Param Array (an array which directly accepts values as parameter rather a reference to an array)
3. Only one param array is possible to be declared as part of the method argument list
4. Any known argument should be passed to the method first and then all the unknown number of arguments should be passed to the param array of the method argument list
5. Consequentially, the method should have known number arguments first and then param array should be the last in the argument list

Jagged Array

- It is an array of arrays
- It is an array containing multiple rows, but each with different number of elements
- DataTable class, used in ADO.NET contains a jagged array to store multiple records, where each record might have different number of attributes (columns)




```
//ask the user to enter no of rows for the jagged array
Console.Write("Enter no of rows for the jagged array: ");
int no_of_rows = Convert.ToInt32(Console.ReadLine());

//jagged array declaration with the specified no of rows
int[][] jaggedarray = new int[no_of_rows][];

for (int row = 0; row < jaggedarray.Length; row++)
{
   //ask user to enter no of elements in a row
   Console.Write("Enter no of elements for jaggedarray["
      + row + "]: ");
   int no_of_elements = Convert.ToInt32(Console.ReadLine());

   //create a single dimensional array with same number of elements
   int[] singlearray = new int[no_of_elements];

   //pass the reference of that single dimensional array to
   //a row of jagged array
   jaggedarray[row] = singlearray;
}
```


Note: Jagged array could be an array of single or multi dimensional arrays as well as even jagged arrays, too.

Entering Items Into and Displaying Items From Jagged Array

```
{
   for (int column = 0; column < jaggedarray[row].Length;
      column++)
   {
      //ask the user to enter item in particular column of
      //particular row
      Console.Write("Enter item in jaggedarray["
         + row + "][" + column + "]: ");
      jaggedarray[row][column] = Convert.ToInt32(
         Console.ReadLine());
   }
   Console.WriteLine("\n");
}

for (int row = 0; row < jaggedarray.Length; row++)
{
   for (int column = 0; column < jaggedarray[row].Length;
      column++)
   {
      //display to the user an item from particular column fror
      //particular row
      Console.Write("Item at jaggedarray["
         + row + "][" + column + "]: "
         + jaggedarray[row][column]);
   }
   Console.WriteLine("\n");
```


Code using Array

ProductArrayRepo.cs

```
class ProductArrayRepo
{
    Product[] products = null;
    readonly int MAX;
    int RecCount = -1;

    1 reference
    public ProductArrayRepo(int max)
    {
        this.MAX = max;
        products = new Product[this.MAX];
    }

    5 references
    public void AddProduct(Product product)
    {
        this.RecCount++;
        this.products[this.RecCount] = product;
    }

    1 reference
    public Product[] GetProducts()
    {
        return this.products;
    }
}
```

ShoponMain.cs

```
public void Main()
{
    ProductArrayRepo productRepo = new ProductArrayRepo(5);
    //1. Store products
    StoreProducts(productRepo);
    //2. Display products
    DisplayProducts(productRepo);
}

1 reference
private void DisplayProducts(ProductArrayRepo productRepo)...

1 reference
private void StoreProducts(ProductArrayRepo productRepo)...
```

Limitation of this approach:
- As end user I will not know how many products I need.
- If we specify the MAX more than our requirements, program will simply occupy space.

Collections

- What?
  - An object that stores a group of elements together as one unit.
  - A group of data of similar type semantically grouped for a purpose.
  - Arrays are the most primitive type of Collections available in all Languages.
- Why?
  - Traditional Arrays are fixed in Size.
  - Real time Applications needs an ability to append, Remove or modify the existing Data as per their needs.
  - An Example without an Idea of Dynamic Data.
- How?
  - .NET framework provides a huge sets of classes which fulfill the issues of Collections.
  - Grouped under the namespace System.Collections.

System.Collections

- The .NET framework has a variety of built in Classes which provide a puzzling selection of collection objects, each with a somewhat specialized purpose.
- You can use the *System.Array* class or the classes in the
  - *System.Collections,*
  - *System.Collections.Generic,*
  - *System.Collections.Concurrent and*
  - *System.Collections.Immutable* namespaces to add, remove, and modify either individual elements or a range of elements in a collection.

https://docs.microsoft.com/en-us/dotnet/standard/collections/

The principal benefit of collections is that they standardize the way groups of objects are handled by our programs. All collections are designed around a set of clearly defined interfaces. Several built-in implementations of these interfaces, such as **ArrayList**, **Hashtable**, **Stack**, and **Queue**, are provided, which we can use as-is. We can also implement our own collection, but we will rarely need to.

- The non-generic collections implement several fundamental data structures, including a dynamic array, stack, and queue. They also include *dictionaries,* in which you can store key/ value pairs. An essential point to understand about the non-generic collections is that they operate on data of type **object**. Thus, they can be used to store any type of data, and different types of data can be mixed within the same collection. Of course, because they store **object** references, they are not type-safe. The non-generic collection classes and interfaces are in **System.Collections**.

- The specialized collections operate on a specific type of data or operate in a unique way. For example, there are specialized collections for strings. There are also specialized collections that use a singly linked list. The specialized collections are declared in **System.Collections.Specialized.**

Collections - Types

- There are two main types of collections;
  1. Generic collections and
  2. Non-generic collections.

Generic Collections

- Generic collections are type-safe at compile time.
- Generic collections typically offer better performance.
- Generic collections accept a type parameter when they are constructed and do not require that we cast to and from the *Object* type when we add or remove items from the collection.
- Generic are type safe way of storing the objects as Data structures
  - Available since v2.0
  - Easy way of storing objects

```
List<int> list = new List<int>();
list.Add(123);//Stores the data as integer.
list.Add("SomeName");/*Does not Accepts this and
Compiler tells about this Error.*/
```


Non-Generic Collections

- Non-generic collections store items as *Object*, require casting, and most are not supported for Windows Store app development.
- Not type safe. Available in v1.0 and v1.1

```
ArrayList list = new ArrayList();
list.Add(123);//Stores the data as object(Boxed Value)
list.Add("SomeName");//Accepts this, but Wrong interpretation.
//Compiler does not throw Error.
```


Choose a collection

I want to…	Generic collection options	Non-generic collection options	Thread-safe or immutable collection options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable	ConcurrentDictionary<TKey,TValue>
		(A collection of key/value pairs that are organized based on the hash code of the key.)	ReadOnlyDictionary<TKey,TValue>
			ImmutableDictionary<TKey,TValue>
Access items by index	List<T>	Array	ImmutableList<T>
		ArrayList	ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T>
			ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T>
			ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TValue>
			ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T>	No recommendation	ImmutableHashSet<T>
	SortedSet<T>		ImmutableSortedSet<T>

https://docs.microsoft.com/en-us/dotnet/standard/collections/

Dictionary

- The *Dictionary<TKey,TValue>* generic class provides a mapping from a set of keys to a set of values. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its **key** is very fast, because the *Dictionary<TKey,TValue>* class is implemented as a hash table.
- The speed of retrieval depends on the quality of the hashing algorithm of the type specified for TKey.
- Every key in a *Dictionary<TKey,TValue>* *must be unique* according to the dictionary's equality comparer.
- A *key cannot be null*, but a value can be, if its type TValue is a reference type.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-6.0

Dictionary – Example1

Initialization of dictionary

```
// Create a new dictionary of strings, with string keys.
//
Dictionary<string, string> openWith =
    new Dictionary<string, string>();

1 reference
public DictionaryDemo1()
{
    // Add some elements to the dictionary. There are no
    // duplicate keys, but some of the values are duplicates.
    openWith.Add("txt", "notepad.exe");
    openWith.Add("bmp", "paint.exe");
    openWith.Add("dib", "paint.exe");
    openWith.Add("rtf", "wordpad.exe");
}
public void Demo1()
{
    // The Add method throws an exception if the new key is
    // already in the dictionary.
    try
    {
        openWith.Add("txt", "winword.exe");
    }
    catch (ArgumentException)
    {
        Console.WriteLine("An element with Key = \"txt\" already exists.");
    }
}
```

Output

```
An element with Key = "txt" already exists.
```


Dictionary – Example2

```
public void Demo2()
{
    // The Item property is another name for the indexer, so you
    // can omit its name when accessing elements.
    Console.WriteLine("For key = \"rtf\", value = {0}.",
        openWith["rtf"]);

    // The indexer can be used to change the value associated
    // with a key.
    openWith["rtf"] = "winword.exe";
    Console.WriteLine("For key = \"rtf\", value = {0}.",
        openWith["rtf"]);

    // If a key does not exist, setting the indexer for that key
    // adds a new key/value pair.
    openWith["doc"] = "winword.exe";

    // The indexer throws an exception if the requested key is
    // not in the dictionary.
    try
    {
        Console.WriteLine("For key = \"tif\", value = {0}.",
            openWith["tif"]);
    }
    catch (KeyNotFoundException)
    {
        Console.WriteLine("Key = \"tif\" is not found.");
    }
}
```

Output

```
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Key = "tif" is not found.
```


Dictionary – Example3

```
public void Demo3()
{
    // When a program often has to try keys that turn out not to
    // be in the dictionary, TryGetValue can be a more efficient
    // way to retrieve values.
    string value = "";
    if (openWith.TryGetValue("tif", out value))
    {
        Console.WriteLine("For key = \"tif\", value = {0}.", value);
    }
    else
    {
        Console.WriteLine("Key = \"tif\" is not found.");
    }

    // ContainsKey can be used to test keys before inserting
    // them.
    if (!openWith.ContainsKey("ht"))
    {
        openWith.Add("ht", "hypertrm.exe");
        Console.WriteLine("Value added for key = \"ht\": {0}",
            openWith["ht"]);
    }
}
```

Output

```
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe
```


Dictionary – Example4

```
public void Demo4()
{
    // When you use foreach to enumerate dictionary elements,
    // the elements are retrieved as KeyValuePair objects.
    Console.WriteLine();
    foreach (KeyValuePair<string, string> kvp in openWith)
    {
        Console.WriteLine("Key = {0}, Value = {1}",
            kvp.Key, kvp.Value);
    }

    // To get the values alone, use the Values property.
    Dictionary<string, string>.ValueCollection valueColl =
        openWith.Values;

    // The elements of the ValueCollection are strongly typed
    // with the type that was specified for dictionary values.
    Console.WriteLine();
    foreach (string s in valueColl)
    {
        Console.WriteLine("Value = {0}", s);
    }
}
```

Output

```
Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = wordpad.exe

Value = notepad.exe
Value = paint.exe
Value = paint.exe
Value = wordpad.exe
```


Dictionary – Example5

```
public void Demo5()
{
    // To get the keys alone, use the Keys property.
    Dictionary<string, string>.KeyCollection keyColl =
        openWith.Keys;

    // The elements of the KeyCollection are strongly typed
    // with the type that was specified for dictionary keys.
    Console.WriteLine();
    foreach (string s in keyColl)
    {
        Console.WriteLine("Key = {0}", s);
    }

    // Use the Remove method to remove a key/value pair.
    Console.WriteLine("\nRemove(\"doc\")");
    openWith.Remove("doc");

    if (!openWith.ContainsKey("doc"))
    {
        Console.WriteLine("Key \"doc\" is not found.");
    }
}
```

Output

```
Key = txt
Key = bmp
Key = dib
Key = rtf

Remove("doc")
Key "doc" is not found.
```


Dictionary – Shopon Implementation

```
class DictionaryCollectionRepo : IProductRepo
{
    Dictionary<int, Product> products = new Dictionary<int, Product>();

    8 references
    public void AddProduct(Product product)
    {
        this.products.Add(product.Pid, product);
    }

    4 references
    public IEnumerable<Product> GetProducts()
    {
        return this.products.Values;
    }

    4 references
    public void RemoveProduct(int id)
    {
        this.products.Remove(id);
    }
}
```


List<T>

- Represents a strongly typed list of objects that can be accessed by index.
- Provides methods to search, sort, and manipulate lists.
- The *List<T>* class is the **generic** equivalent of the *ArrayList* class. It implements the *IList<T>* generic interface by using an array whose size is dynamically increased as required.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0

List<T> Example

```
public void Main()
{
    List<Student> students = new List<Student>()
    {
        new Student(){Sid = 6, Age = 22, Name = "Rama"},
        new Student(){Sid = 2, Age = 21, Name = "Sita"},
        new Student(){Sid = 1, Age = 23, Name = "Geeta"},
        new Student(){Sid = 3, Age = 21, Name = "Raju"},
        new Student(){Sid = 5, Age = 22, Name = "Anu"},
    };

    //1. Display students details
    DisplayStudents(students);

    //2. Compare Students
    CompareStudents();
```

```
private void DisplayStudents(List<Student> students)...
private void DisplayStudents(List<Student> students)
{
    Console.WriteLine("ID\tAge\tName");
    Console.WriteLine("---------------------------");
    foreach (var student in students)
    {
        Console.WriteLine($"{student.Sid}" +
            $"\t{student.Age}" +
            $"\t{student.Name}");
    }
    Console.WriteLine("---------------------------");
}
```

```
private void CompareStudents()
{
    Student student1 = new Student() { Sid = 1001, Name = "Ravi", Age = 21 };
    Student student2 = new Student() { Sid = 1001, Name = "Ravi", Age = 21 };
    if (student1.Equals(student2))
    {
        Console.WriteLine("Both are equal");
    }
    else
    {
        Console.WriteLine("Not equal");
    }

    if (student1 == student2)
    {
        Console.WriteLine("Both are equal");
    }
    else
    {
        Console.WriteLine("Not equal");
    }
}
```


List<T> Example (continued)

```
    //3. Insert new student
    Console.WriteLine("After inserting new students");
    InsertStudentAtIndex(4, new Student() { Sid = 4, Name = "Nancy", Age = 21 }, students);
    DisplayStudents(students);
    //4. Insert Students
    InsertStudents(students);
    Console.WriteLine("After inserting new set of students");
    DisplayStudents(students);
    //5. Remove Student
    RemoveStudent(5, students);
    Console.WriteLine($"After deleting student with id {5}");
    DisplayStudents(students);
}
```

```
private void InsertStudentAtIndex(int index,
    Student student,
    List<Student> students)
{
    students.Insert(index, student);
}

private void InsertStudents(List<Student> students)
{
    List<Student> newStudents = new List<Student>()
    {
        new Student(){Sid = 7, Name = "Anju", Age = 21},
        new Student(){Sid = 8, Name = "Bharath", Age = 23},
        new Student(){Sid = 9, Name = "Chitra", Age = 21},
    };
    students.AddRange(newStudents);
}

private void RemoveStudent(int id, List<Student> students)
{
    var student = students.FirstOrDefault(x => x.Sid == id);
    if(student == null)
    {
        Console.WriteLine($"No student with {id} found.");
    }
    else
    {
        students.Remove(student);
        Console.WriteLine("Student deleted.");
    }
}
```


Queue<T>

- Represents a first-in, first-out collection of objects.
- Objects stored in a Queue are inserted at one end and removed from the other.
- Use Queue if we need to access the information in the same order that it is stored in the collection.
- Operations
  - **Enqueue** adds an element to the end of the Queue.
  - **Dequeue** removes the oldest element from the start of the Queue.
  - **Peek** returns the oldest element that is at the start of the Queue but does not remove it from the Queue.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue?view=net-6.0

Queue<T> - Example

```
public void Main()
{
    Queue<string> names = new Queue<string>();
    names.Enqueue("Rana");
    names.Enqueue("Ravi");
    names.Enqueue("Raju");
    names.Enqueue("Ramya");
    names.Enqueue("Suman");
    DisplayNames(names);
    Console.WriteLine($"Name removed is {names.Dequeue()}");
    Console.WriteLine($"Name removed is {names.Dequeue()}");
    Console.WriteLine("After removing names");
    Console.WriteLine("----------------------");
    DisplayNames(names);
}

private void DisplayNames(Queue<string> names)
{
    foreach (var name in names)
    {
        Console.WriteLine(name);
    }
}
```

Output

```
Rana
Ravi
Raju
Ramya
Suman
Name removed is Rana
Name removed is Ravi
After removing names
----------------------
Raju
Ramya
Suman
```


Stack<T>

- Represents a simple last-in-first-out (LIFO) non-generic collection of objects.
- Stack accepts null as a valid value and allows duplicate elements.
- **Push** is used to add new element. **Pop** is used to remove the element.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack?view=net-6.0

Stack<T> - Example

```
public void Main()
{
    Stack<string> names = new Stack<string>();
    names.Push("Raju");
    names.Push("Anju");
    names.Push("Sanju");
    names.Push("Manju");
    names.Push("Raman");
    DisplayNames(names);
    //Remove names
    names.Pop();
    names.Pop();
    Console.WriteLine("--------------------");
    Console.WriteLine("After removing names");
    Console.WriteLine("--------------------");
    DisplayNames(names);
}

private void DisplayNames(Stack<string> names)
{
    foreach (var name in names)
    {
        Console.WriteLine(name);
    }
}
```

Output

```
Raman
Manju
Sanju
Anju
Raju
--------------------
After removing names
--------------------
Sanju
Anju
Raju
```


LinkedList<T>

- Represents a doubly linked list.
- **LinkedList<T>** is a general-purpose linked list. It supports enumerators and implements the **ICollection** interface.
- Each node in a **LinkedList<T>** object is of the type **LinkedListNode<T>**. Because the **LinkedList<T>** is doubly linked, each node points forward to the **Next** node and backward to the **Previous** node.
- Lists that contain reference types perform better when a node and its value are created at the same time.
- **LinkedList<T>** accepts null as a valid **Value** property for reference types and allows duplicate values.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-6.0

LinkedList<T> - Example

```
public void Main()
{
    string[] names = { "Anju","Anu","Chitra","Geetha" };
    LinkedList<string> friends = new LinkedList<string>(names);
    //1. Display all friends
    Display(friends);
    Console.WriteLine("--------------------------");
    //2. Add friends
    AddFriends(friends);
    //3. Remove friend
    RemoveFriends(friends);
}
private void AddFriends(LinkedList<string> friends)
{
    friends.AddFirst("Shiva");
    Console.WriteLine("After add Shiva at First");
    Console.WriteLine("--------------------------");
    Display(friends);
    Console.WriteLine("--------------------------");
    LinkedListNode<string> existingNode = friends.FindLast("Anu");
    friends.AddAfter(existingNode, "Bhaskar");
    Console.WriteLine("Adding Bhaskar after Anu");
    Console.WriteLine("--------------------------");
    Display(friends);
    Console.WriteLine("--------------------------");
    Console.WriteLine("Adding Ramu at the end");
    Console.WriteLine("--------------------------");
    friends.AddLast("Ramu");
    Display(friends);
    Console.WriteLine("--------------------------");
}
```

```
private void Display(LinkedList<string> friends)
{
    foreach (var friend in friends)
    {
        Console.WriteLine(friend);
    }
}
```

```
private void RemoveFriends(LinkedList<string> friends)
{
    string removeFriend = "Shiva";
    friends.Remove(removeFriend);
    Console.WriteLine("After removing node - Shiva");
    Console.WriteLine("---------------------------");
    Display(friends);
    Console.WriteLine("---------------------------");
}
```


ObservableCollection<T>

- Represents a dynamic data collection that provides notifications when items get added, removed, or when the whole list is refreshed.
- To set up dynamic bindings to the UI component so that insertions or deletions in the collection update the UI automatically, the collection must implement the **INotifyCollectionChanges** interface.
- This interface exposes the **CollectionChanged** event, an event that should be raised whenever the underlying collection changes.
- Found in the namespace **System.Collections.ObjectModel**

https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.observablecollection-1?view=net-6.0#definition

ObservableCollection<T> - Example

```
class CartItem
{
    6 references
    public int Pid { get; set; }
    7 references
    public double Price { get; set; }
    8 references
    public int Qty { get; set; }
}

class ObservableCollectionDemo
{
    private ObservableCollection<CartItem> cartItems =
            new ObservableCollection<CartItem>();
    private double totalAmount = 0;
    private int itemCount = 0;
    1 reference
    public void Main()...

    1 reference
    private void RemoveItem()...

    2 references
    private void DisplayCartItems()...
```

```
public void Main()
{
    //1. Add Event to handle cart
    cartItems.CollectionChanged += CartItems_CollectionChanged;
    //2. Initialize with test data
    InitCartItem();
    //3. Display all products
    DisplayCartItems();
    //4. Remove item from cart
    RemoveItem();
    Console.WriteLine("After removing item");
    DisplayCartItems();
}

private void RemoveItem()
{
    cartItems.RemoveAt(3);
}

private void DisplayCartItems()
{
    Console.WriteLine("Pid \t Price \t Qty");
    Console.WriteLine("----------------------------");
    foreach (var item in this.cartItems)
    {
        Console.WriteLine($"{item.Pid} \t {item.Price} " +
            $"\t {item.Qty}");
    }
    Console.WriteLine("----------------------------");
    Console.WriteLine($"Total Amount :{this.totalAmount} " +
            $"\t Item Count :{this.itemCount}");
    Console.WriteLine("----------------------------");
}
```


ObservableCollection<T> - Example (continued)

```
    1 reference
    private void CartItems_CollectionChanged(object sender,
            NotifyCollectionChangedEventArgs e)...

    1 reference
    private void FindSumOfItems()...

    1 reference
    public void InitCartItem()...
}
```

```
private void CartItems_CollectionChanged(object sender,
        NotifyCollectionChangedEventArgs e)
{
    if(e.Action == NotifyCollectionChangedAction.Add ||
        e.Action == NotifyCollectionChangedAction.Remove)
    {
        FindSumOfItems();
    }
}
private void FindSumOfItems()
{
    this.totalAmount = 0;
    this.itemCount = 0;

    foreach (var item in this.cartItems)
    {
        this.totalAmount += (item.Price * item.Qty);
        this.itemCount += item.Qty;
    }
}

public void InitCartItem()
{
    var testData = new ObservableCollection<CartItem>()
    {
        new CartItem(){Pid = 1001, Price = 12000, Qty = 2},
        new CartItem(){Pid = 1002, Price = 11000, Qty = 1},
        new CartItem(){Pid = 1003, Price = 9900, Qty = 1},
        new CartItem(){Pid = 1004, Price = 10500, Qty = 3},
        new CartItem(){Pid = 1005, Price = 13000, Qty = 2}
    };

    //add items to cart
    foreach (var item in testData)
    {
        this.cartItems.Add(item);
    }
}
```


SortedList

- SortedList<TKey,TValue> a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.
- A SortedList element can be accessed by its key, like an element in any IDictionary implementation, or by its index, like an element in any IList implementation.
- A key cannot be null, but a value can be.
- The elements of a SortedList object are sorted by the keys either according to a specific IComparer implementation specified when the SortedList is created or according to the IComparable implementation provided by the keys themselves. In either case, a SortedList does not allow duplicate keys.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist?view=net-6.0

SortedList - Example

```
SortedList<int, string> contacts = null;
1 reference
public SortedListDemo()
{
    contacts = new SortedList<int, string>();
    contacts.Add(1005, "Shiva");
    contacts.Add(1002, "Chitra");
    contacts.Add(1001, "Sham");
    contacts.Add(1004, "Bharath");
    contacts.Add(1003, "Anju");
}

public void Main()
{
    //1. Display contacts
    DisplayContacts();
    //2. Sort by name
    SortByName();
}

1 reference
private void SortByName()
{
    var sortedData = this.contacts.OrderBy(x => x.Value);
    Console.WriteLine("Sorted by name");
    Console.WriteLine("-------------");
    foreach (var contact in sortedData)
    {
        Console.WriteLine($"{contact.Key}\t{contact.Value}");
    }
}
```

```
private void DisplayContacts()
{
    foreach (var contact in this.contacts)
    {
        Console.WriteLine($"{contact.Key}\t{contact.Value}");
    }
}
```

Output

```
1001    Sham
1002    Chitra
1003    Anju
1004    Bharath
1005    Shiva
Sorted by name
-------------
1003    Anju
1004    Bharath
1002    Chitra
1001    Sham
1005    Shiva
```


HashSet<T>

- The HashSet<T> class provides high-performance set operations. A set is a collection that contains no duplicate elements, and whose elements are in no particular order.
- A HashSet<T> collection is not sorted and cannot contain duplicate elements. If order or element duplication is more important than performance for our application, consider using the List<T> class together with the Sort method.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-6.0

HashSet<T> - Example

```
HashSet<string> Boys;
HashSet<string> Girls;

1 reference
public HashsetDemo()
{
    this.Boys = new HashSet<string>()
    {
        "Shiva", "Shankar", "Ravi", "Suman"
    };
    this.Girls = new HashSet<string>()
    {
        "Geetha", "Radha", "Suma"
    };
}
public void Main()
{
    //1. Display Boys and Girls
    Console.WriteLine("Boys List");
    Console.WriteLine("---------");
    Display(this.Boys);
    Console.WriteLine("\nGirls List");
    Console.WriteLine("---------");
    Display(this.Girls);
    //2. Combine the list
    var combined = this.Boys.Union(Girls);
    Console.WriteLine("\nAfter combining");
    Console.WriteLine("---------");
    Display(combined);
}
```

```
private void Display(IEnumerable<string> data)
{
    foreach (var item in data)
    {
        Console.WriteLine(item);
    }
}
```

Output

```
Boys List
---------
Shiva
Shankar
Ravi
Suman

Girls List
---------
Geetha
Radha
Suma

After combining
---------
Shiva
Shankar
Ravi
Suman
Geetha
Radha
Suma
```


SortedSet<T>

- Represents a collection of objects that is maintained in sorted order.
- A **SortedSet<T>** object maintains a sorted order without affecting performance as elements are inserted and deleted.
- Duplicate elements are not allowed. Changing the sort values of existing items is not supported and may lead to unexpected behavior.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1?view=net-6.0

SortedSet<T> - Example

```
SortedSet<string> Boys;
SortedSet<string> Girls;

1 reference
public SortedSetDemo()
{
    this.Boys = new SortedSet<string>()
    {
        "Ravi","Bhanu","Chandra","Saheer"
    };
    this.Girls = new SortedSet<string>()
    {
        "Bhavana","Deepa","Suma","Anju"
    };
}
public void Main()
{
    Console.WriteLine("Boys List");
    Console.WriteLine("---------");
    Display(this.Boys);
    Console.WriteLine("\nGirls List");
    Console.WriteLine("---------");
    Display(this.Girls);
    //2. Combine the list
    var combined = this.Boys.Union(Girls);
    Console.WriteLine("\nAfter combining");
    Console.WriteLine("---------");
    Display(combined);
}
```

```
private void Display(IEnumerable<string> data)
{
    foreach (var item in data)
    {
        Console.WriteLine(item);
    }
}
```

Output

```
Boys List
---------
Bhanu
Chandra
Ravi
Saheer

Girls List
---------
Anju
Bhavana
Deepa
Suma

After combining
---------
Bhanu
Chandra
Ravi
Saheer
Anju
Bhavana
Deepa
Suma
```


ConcurrentDictionary

- ConcurrentDictionary<TKey,TValue> represents a thread-safe collection of key/value pairs that can be accessed by multiple threads concurrently.
- Like the System.Collections.Generic.Dictionary<Tkey,Tvalue> class, ConcurrentDictionary<Tkey, Tvalue> implements the IDictionary<Tkey, Tvalue> interface.
- Present in System.Collections.Concurrent namespace

https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=net-6.0

ConcurrentDictionary - Example

```
//      ConcurrentDictionary<TKey, TValue> ctor(concurrencyLevel, initialCapacity)
1 reference
public void Main()
{
    // We know how many items we want to insert into the ConcurrentDictionary.
    // So set the initial capacity to some prime number above that, to ensure that
    // the ConcurrentDictionary does not need to be resized while initializing it.
    int NUMITEMS = 64;
    int initialCapacity = 101;

    // The higher the concurrencyLevel, the higher the theoretical number of operations
    // that could be performed concurrently on the ConcurrentDictionary.  However, global
    // operations like resizing the dictionary take longer as the concurrencyLevel rises.
    // For the purposes of this example, we'll compromise at numCores * 2.
    int numProcs = Environment.ProcessorCount;
    int concurrencyLevel = numProcs * 2;

    // Construct the dictionary with the desired concurrencyLevel and initialCapacity
    ConcurrentDictionary<int, int> cd = new ConcurrentDictionary<int, int>(concurrencyLevel, initialCapacity);
    // Initialize the dictionary
    for (int i = 0; i < NUMITEMS; i++) cd[i] = i * i;

    Console.WriteLine("The square of 23 is {0} (should be {1})", cd[23], 23 * 23);
}
```

Output

```
The square of 23 is 529 (should be 529)
```


Interfaces

- The .NET Framework provides standard interfaces for enumerating, comparing, and creating collections. The key collection interfaces are

Interface	Purpose
IEnumerable	Enumerates through a collection using a foreach statement.
IEnumerator	**IEnumerator** is an interface which helps to get current elements from the collection.
IComparable	Defines a generalized type-specific comparison method that a value type or class implements to order or sort its instances.
IComparer	Compares two objects held in a collection so that the collection can be sorted.
IDictionary	For key/value-based collections such as Hashtable and SortedList.
IDictionaryEnumerator	Allows enumeration with foreach of a collection that supports IDictionary.
ICollection	Implemented by all collections to provide the CopyTo( ) method as well as the Count, ISReadOnly, ISSynchronized, and SyncRoot properties.

IEnumerable

- Exposes an enumerator, which supports a simple iteration over a non-generic collection.
- It is the best practice for iterating a custom collection by implementing the IEnumerable and IEnumerator interfaces.
- IEnumerable contains a single method, GetEnumerator, which returns an IEnumerator. IEnumerator provides the ability to iterate through the collection by exposing a Current property and MoveNext and Reset methods.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=net-6.0

IEnumerator

- IEnumerator is an interface which helps to get current elements from the collection, it has the following two methods
  - MoveNext() - Sets the enumerator to the next element of the collection; it Returns true if the enumerator was successfully set to the next element and false if the enumerator has reached the end of the collection.
  - Reset() - Sets the enumerator to its initial position, which is before the first element in the collection.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=net-6.0

IEnumerable vs IEnumerator

- While reading these two names, it can be confusing, so let us understand the difference between these two.
  - IEnumerable and IEnumerator are both interfaces.
  - IEnumerable has just one method called GetEnumerator. This method returns another type which is an interface that interface is IEnumerator.
  - If we want to implement enumerator logic in any collection class, it needs to implement IEnumerable interface (either generic or non-generic).
  - IEnumerable has just one method whereas IEnumerator has two methods (MoveNext and Reset) and a property Current.
- For our understanding, we can say that IEnumebale is a box that contains IEnumerator inside it.

IComparable<T>

- Defines a generalized type-specific comparison method that a value type or class implements to *order* or *sort* its instances.
- This interface is implemented by types whose values can be ordered or sorted. **CompareTo(Object)** indicates whether the position of the current instance in the sort order is before, after, or the same as a second object of the same type.
- The instance's **IComparable** implementation is called automatically by methods such as **Array.Sort** and **ArrayList.Sort.**

Value	Meaning
Less than zero	The current instance precedes the object specified by the CompareTo method in the sort order.
Zero	This current instance occurs in the same position in the sort order as the object specified by the CompareTo method.
Greater than zero	This current instance follows the object specified by the CompareTo method in the sort order.

https://docs.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-6.0

IComparable<T> - Example

```
public class Product : IEquatable<Product>, IComparable<Product>
{
    23 references
    public int Pid { get; set; }
    14 references
    public string ProductName { get; set; }
    14 references
    public double Price { get; set; }
    14 references
    public string AvailableStatus { get; set; }
    14 references
    public string ImageUrl { get; set; }
    public int CompareTo(Product other)
    {
        if (other == null) return 1;
        if(this.Pid > other.Pid)
        {
            return 1;
        }
        else if(this.Pid < other.Pid)
        {
            return -1;
        }
        else
        {
            return 0;
        }
    }
    2 references
    public bool Equals(Product other)
    {
        if (other == null) return false;
        return this.Pid == other.Pid;
    }
}
```

Output

```
Sort by PID
-----------------------------------
Pid     Name          Price   Image URL              Status
-------------------------------------------------------------------
1001    Apple I7      13000   images/applei7.jpg     Y
1002    Nokia X       25000   images/nokiax.jpg      Y
1003    Note J        23000   images/notej.jpg       Y
1004    Note S        23000   images/notes.jpg       Y
1005    Honor YT      26000   images/honoryt.jpg     Y
```


IComparer<T>

- Exposes a method that compares two objects.
- This interface is used in conjunction with the **Array.Sort** and **Array.BinarySearch** methods.
- It provides a way to customize the sort order of a collection.

https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer?view=net-6.0

IComparer<T> - Example

ProductNameSorter.cs

```
public class ProductNameSorter : IComparer<Product>
{
    0 references
    public int Compare(Product x, Product y)
    {
        return x.ProductName.CompareTo(y.ProductName);
    }
}
```

This is new class which will be used while sorting the data based on the Compare logic.

ProductMain.cs

```
private void SortByProductName()
{
    var temp = manager.GetProducts().ToList();
    temp.Sort(new ProductNameSorter());
    Console.WriteLine("\nSort by Product Name");
    Console.WriteLine("-------------------------------------");
    DisplayProducts(temp);
}
```

Output

```
Sort by Product Name
-------------------------------------
Pid     Name            Price   Image URL               Status
-----------------------------------------------------------------
1001    Apple I7        13000   images/applei7.jpg      Y
1005    Honor YT        26000   images/honoryt.jpg      Y
1002    Nokia X         25000   images/nokiax.jpg       Y
1003    Note J          23000   images/notej.jpg        Y
1004    Note S          23000   images/notes.jpg        Y
```


Indexers

- C# indexers are usually known as smart arrays.
- A C# indexer is a class property that allows you to access a member variable of a class or struct using the features of an array.
- The indexed value can be set or retrieved without explicitly specifying a type or instance member.
- Indexers resemble properties except that their accessors take parameters.
- Indexers are created using ***this*** keyword.

https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/

Creating Indexers

```
<modifier> <return type> this [argument list]
{
get
{
// your get block code
}
set
{
// your set block code
}
}
```

<modifier>
can be private, public, protected or internal.

<return type>
can be any valid C# types.

this
this is a special keyword in C# to indicate the object of the current class.

[argument list]
The formal-argument-list specifies the parameters of the indexer.

Points to remember

- Indexers are always created with **this** keyword.
- Parameterized property are called indexer.
- Indexers are implemented through get and set accessors for the [ ] operator.
- ref and out parameter modifiers are not permitted in indexer.
- The formal parameter list of an indexer corresponds to that of a method and at least one parameter should be specified.
- Indexer is an instance member so can't be static but property can be static.
- Indexers are used on group of elements.
- Indexer is identified by its signature where as a property is identified by it's name.
- Indexers are accessed using indexes where as properties are accessed by names.
- Indexer can be overloaded.

Indexers Example

```
class IndexersClass
{
    private string[] names = new string[10];
    11 references
    public string this[int i]
    {
        get
        {
            return names[i];
        }
        set
        {
            names[i] = value;
        }
    }
}
```

```
class IndexersDemo
{
    0 references
    static void Main()
    {
        IndexersClass nameClass = new IndexersClass();
        nameClass[0] = "Shashi";
        nameClass[1] = "Ravi";
        nameClass[2] = "Shankar";
        nameClass[3] = "Somesh";
        nameClass[4] = "Manish";
        nameClass[5] = "Radha";
        nameClass[6] = "Krishna";
        nameClass[7] = "Manju";
        nameClass[8] = "Chandra";
        nameClass[9] = "Saheer";

        for (int i=0; i<10; i++)
        {
            Console.WriteLine(nameClass[i]);
        }
    }
}
```


Next Step

Recap

Useful links

Slide 81

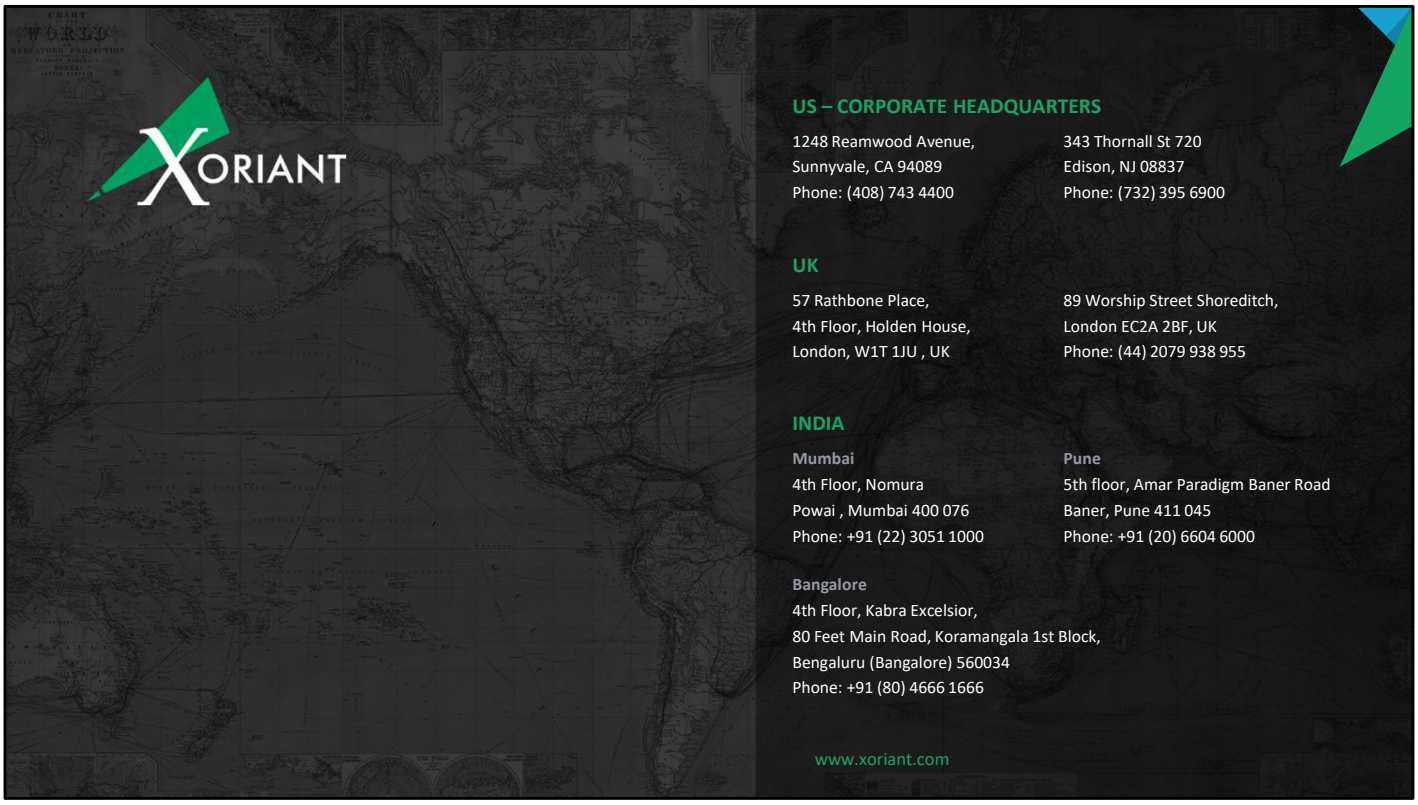
BU0 remove home and add previous button
Bangalore User2, 2022-01-06T04:57:13.599

Recap

- Till now we have understood
  - What are array
  - Limitations of array
  - Collections
  - Types of collections
    - Generic
    - Non-generic
  - Useful interfaces
  - Indexers

Useful Links

- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/
- https://docs.microsoft.com/en-us/dotnet/standard/collections/
- https://docs.microsoft.com/en-us/dotnet/standard/collections/
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.dictionary-2?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.queue?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.observablecollection-1?view=net-6.0#definition
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.sortedlist?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.hashset-1?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sortedset-1?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent.concurrentdictionary-2?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerator?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.icomparable?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/api/system.collections.icomparer?view=net-6.0
- https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/indexers/


XORIANT
US – CORPORATE HEADQUARTERS
1248 Reamwood Avenue,
Sunnyvale, CA 94089
Phone: (408) 743 4400
343 Thornall St 720
Edison, NJ 08837
Phone: (732) 395 6900
UK
57 Rathbone Place,
4th Floor, Holden House,
London, W1T 1JU , UK
89 Worship Street Shoreditch,
London EC2A 2BF, UK
Phone: (44) 2079 938 955
INDIA
Mumbai
4th Floor, Nomura
Powai , Mumbai 400 076
Phone: +91 (22) 3051 1000
Pune
5th floor, Amar Paradigm Baner Road
Baner, Pune 411 045
Phone: +91 (20) 6604 6000
Bangalore
4th Floor, Kabra Excelsior,
80 Feet Main Road, Koramangala 1st Block,
Bengaluru (Bangalore) 560034
Phone: +91 (80) 4666 1666
www.xoriant.com