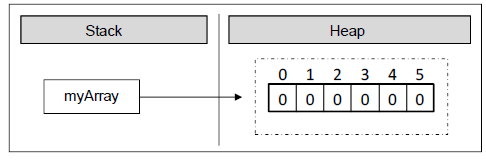
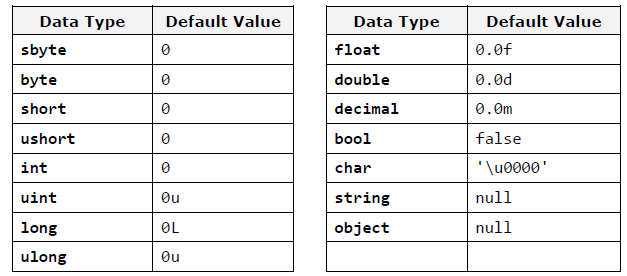
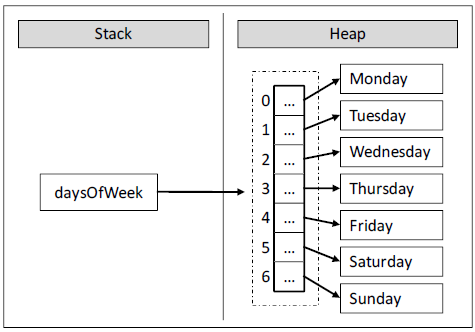
**//------------------------------------------------------------------------- Arrays ---------------------------------------------------------------------//**

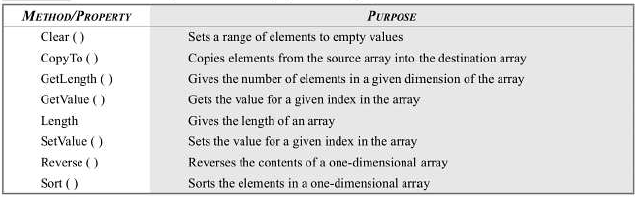
//Array - ordered sequence(set of contiguous data points) of elements/variables of the same type (primitive or reference)  
//This list is given a single variable name - but each element is accessed by a numerical index/subscript starting from 0(called lower bound)  
// 1-dimensional arrays - vectors ; 2-dimensional arrays – matrices  
int[] marks; // [] indicates we are declaring an array of int type elements, not a single element  
string[] names; // On declaration, we create a variable with the identifier marks, names in the **program's execution stack**  
float[] average, percentile; // No memory is allocated yet; declaration only creates a reference which doesn't have a value(points to null)  
//In above lines, we declared *int array reference, string array reference*, etc.  
marks = new int[3]; // We *create array* with the '*new*' keyword - allocates area for 3 integers in the **dynamic memory(heap),**indexed 0 to N-1  
names = new string[5];// Array creation automatically initializes the array elements to a default value, in case user didn't initialize it  
// For bool array, default value is false ; for int array, it is set to 0  
Console.WriteLine(marks[2] + " " + names[3] ); // Outputs: 0 ; string is initialized to null, not seen on output   
if ( names[4] == null )  
{  
 Console.WriteLine("The value is null"); // Outputs: The value if null ; this is how we check  
}  


string[] locations = new string[4]; // Declaration and Creation can be combined into one step  
marks[0] = 4; // Initialization can be done elementwise by accessing each index  
marks[1] = 9; // Trying to access an array beyond its bound will generate error   
marks[2] = -3; // We can access given elements of the array both for reading and for writing, which means we can treat elements as variables.  
  
//marks[**3**] = 10; --> The .NET Framework does automatic check; outputs : System.IndexOutOfRangeException: 'Index was outside the bounds of the array.’  
// Initialization: Specify each array item within the scope of curly brackets ({}) - helpful when creating an array of known size quickly specify the initial values  
string[] wizards = new string[6] { "Harry", "Potter", "Ginny", "Hermione", "Ronald", "Yennefer" };  
// Declaration, Creation, Initialization all in same step  
// Need not specify the size of the array within [], when using curly bracket syntax to specify elements as length of array is auto inferred by number of items within the scope of the curly brackets  
average = new float[] { 4.6f ,7.8f ,9.2f }; // Creation and Initialization can be done in one step   
bool[] states = { true, false, false, true, true };  
string[] daysOfWeek = { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"};  
// Use of the new keyword is optional  
/\* Strings in C# are immutable reference types, unlike integer which are value types. Array of integers holds the int value but array of string holds the reference to the elements and not the actual values. In above case we allocate an array of seven elements of type string. The type string is a reference type (object) and its values are stored in the dynamic memory. The variable daysOfWeek is allocated in the stack memory, and points to a section of the dynamic memory containing the elements of the array. The type of each of these seven elements is string, which itself points to a different section of the dynamic memory, where the real value is stored.\*/

  
// If there is mismatch b/w the declared size within[] & the no. of initializers (too many or too few ), we are issued a compile-time error.  
var prices = new[] { 345.6, 627.3, 419, 215.6 };// Define implicitly typed local arrays using ‘var’, like implicitly typed local variables  
// Here we allocate a new array variable without specifying the type contained within array itself(note we must use the new keyword in this approach).  
// Compiler auto determines underlying type; Items in array’s initialization list must have **same** underlying type; Mixed types generate compile time error  
var greetings = new[] { "Hello", null, "Namaste" }; // prices is really double, greetings is string  
Console.WriteLine("prices is a: {0}", prices.ToString()); // Outputs: prices is a System.Double[]  
// object.ToString returns a string that represents the current object;

// We can provide values to an array by reading it from the console ; loop over array to initialize every element  
Console.WriteLine("Reading input values from user to create an array & check whether it is symmetric.");  
Console.Write("Enter a Positive Integer for the length of the Array:");  
int length = int.Parse(Console.ReadLine()); // int.Parse() : converts string representation of a number to its integer equivalent.  
// If the string does not represent a valid integer, it throws a FormatException.  
int[] array = new int[length];  
Console.WriteLine("Enter the Values of the Array, one by one in separate lines:");  
for(int i =0; i < length; i++)  
{  
 array[i] = int.Parse(Console.ReadLine());  
}  
bool symmetric = true;  
for (int i = 0; i < array.Length / 2 ; i++) // Iterate through elements of an array using a for loop   
{  
 if (array[i] != array[length - i - 1])   
 {  
 symmetric = false;  
 break; // exit prematurely ‘for’ loop once condition is satisfied, transfer control to the statement that follows the terminated statement.  
 }  
}  
Console.WriteLine("Is the Array Symmetric: {0} ", symmetric);  
// An array cannot be directly printed, rather its elements have to be accessed separately  
Console.WriteLine(array); // This outputs only the type of array, not its contents: System.Int32[]

// The 'foreach' construct is different from 'for' loop: iteration is made always through all elements – from the start to the end.  
// We cannot access the current index, we are just iterating through the collection in a way, defined by the collection itself.  
// The loop variable in foreach-loops is read-only so we cannot modify the current loop item from the loop body.  
// foreach (var item in collection) { ...code... } : var is the type of elements we iterate through ; collection is the array or any other collection; item is the current element on each step  
string[] places = { "Bali", "Goa", "Andaman", "Seychelles" };  
foreach(var item in places) // also, foreach(string place in places)   
{  
 Console.Write(item + " "); // Outputs: Bali Goa Andaman Seychelles  
}  
  
// Multi-Dimensional Array - Arrays of Arrays ; A single array- a contiguous line of elements is a vector, each element within this array may itself me another array, thus 2D array. In theory, there is no limit on array dimensions. Declaration: put commas within []  
int[,] intMatrix = new int[3,4]; // There are 3 elements of type int, and each of the elements has a length 4. Visualize as 3 rows and 4 cols.  
string[,,] stringMatrix = new string[2, 3, 4];// 2 string elements, each has 3 elements, within which each has length 4 (cuboid matrix)  
int[,] matrix =  
{  
 { 1, 2, 3, 4 },  
 { 5, 6, 7, 8 }  
};  
Console.WriteLine(matrix[1, 2]); // Outputs: 7 ; All dimensions are indexed from zero; each element is accessed by the format [row,col]  
Console.WriteLine(matrix.GetLength(1)); // Outputs : 4 ; number of the rows of 2D array by using matrix.GetLength(0) and the number of all columns per row with matrix.GetLength(1)  
// Use nested loops to print the whole 2D array by accessing all elements  
// Jagged Arrays: Arrays of arrays, in which array in each row can have length different than those in the other rows.  
// Declaration: With the jagged arrays we have a pair brackets per dimension  
int[][] jaggedArray; // Declaration  
jaggedArray = new int[2][]; // Since columns are not same, only specify the first dimension/no. of rows in initialization  
jaggedArray[0] = new int[3]; // Each row in now again initialized to the number of elements  
jaggedArray[1] = new int[4];

// In C#, all arrays are class based; any array we create is automatically derived from **System.Array** class, which defines many methods and properties to efficiently manipulate arrays; these are accessed using the dot operator.  
  
Console.WriteLine("Wizards array has {0} names.", wizards.Length); // Outputs Wizards array has 6 names