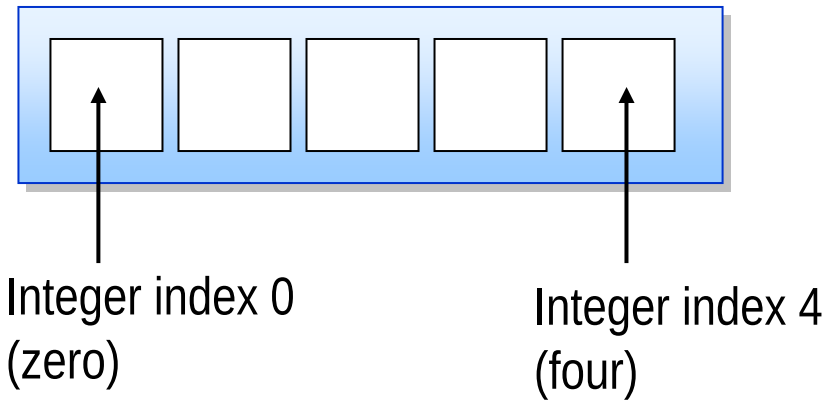
Arrays

Dr. Kwaiter Ghassan

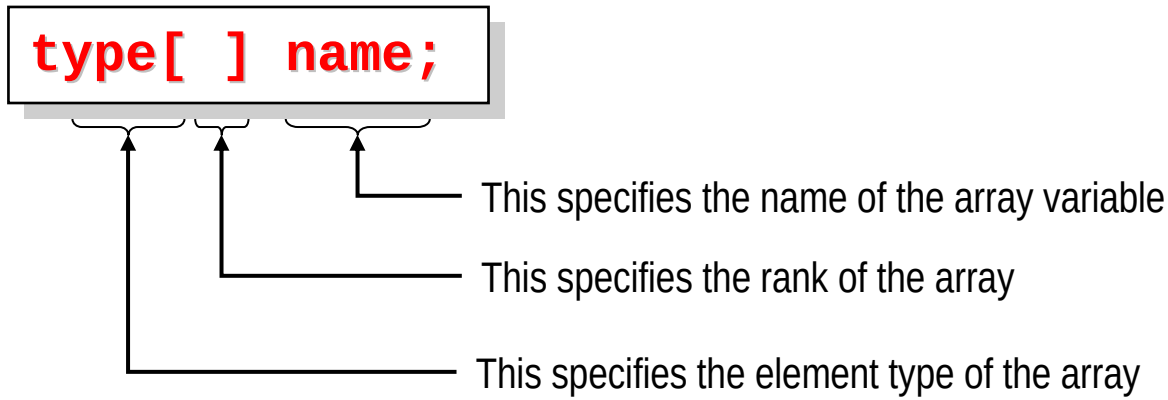
What Is an Array?

- An array is a sequence of elements
  - All elements in an array have the same type
  - Individual elements are accessed using integer indexes



Declaring Arrays

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



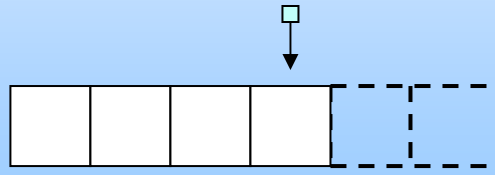
When we declare an array, we do not specify its size.

Array Rank

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

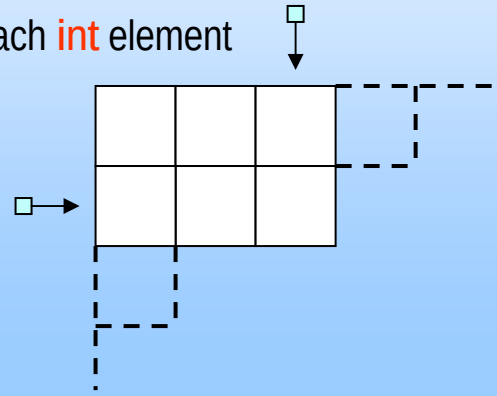
```
int[ ] row;
```

Rank 1: One-dimensional
Single index associates with each int element



```
int[,] grid;
```

Rank 2: Two-dimensional
Two indexes associate with each int element

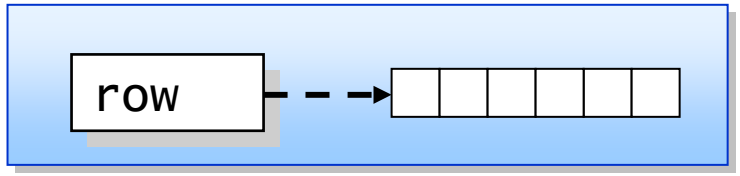


The square brackets [ ] tell the C# compiler that you are declaring an array, and the type specifies the type of the elements it will contain.

Creating Arrays -1

- Instantiate an array using the **new** keyword.

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



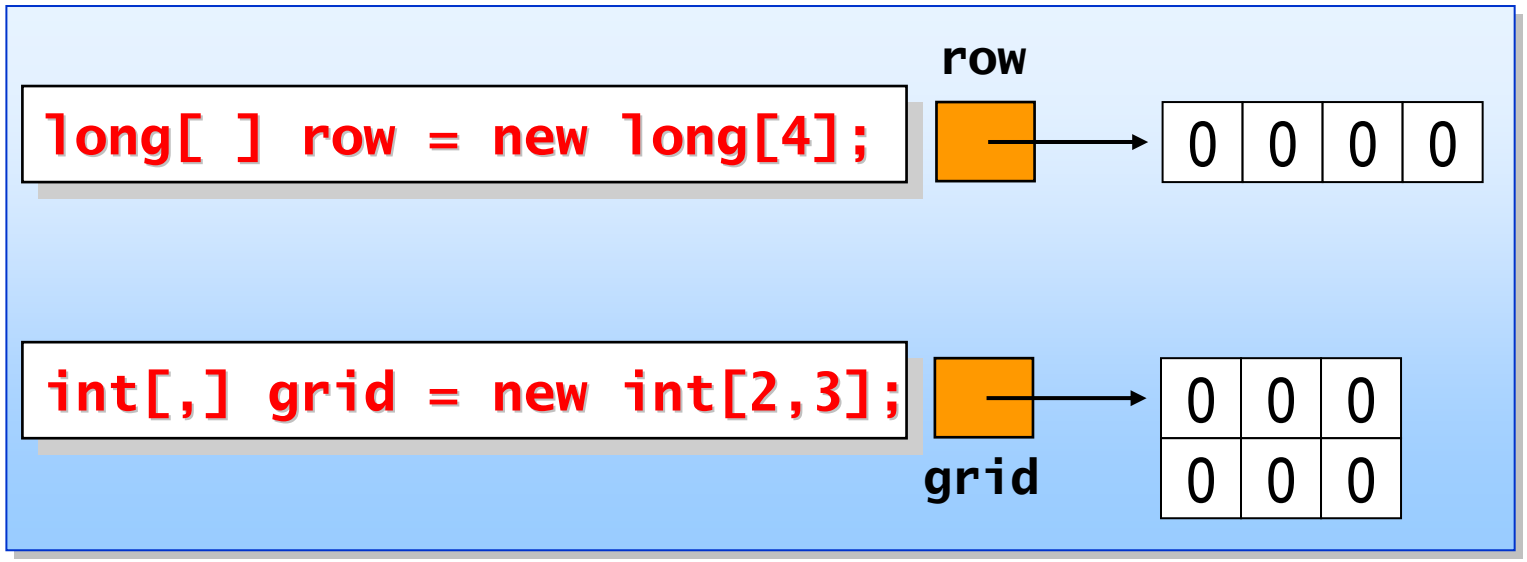
- C# arrays are reference types, created on the heap. Thus, **row** is allocated on the heap.
- The elements of an array are allocated based on their type.
- Integers are value types, and so the elements in **row** will be value types.

Creating Arrays -2

Example

- int [ ] table1;
- char [ ] table2;
- float [ ] table3;
- string [ ] tableStr;

- int [ ] table1 = new int [5];
- char [ ] table2 = new char [12];
- float [ ] table3 = new float [8];
- string [ ] tableStr = new String [9];



Creating Arrays -3

- 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:

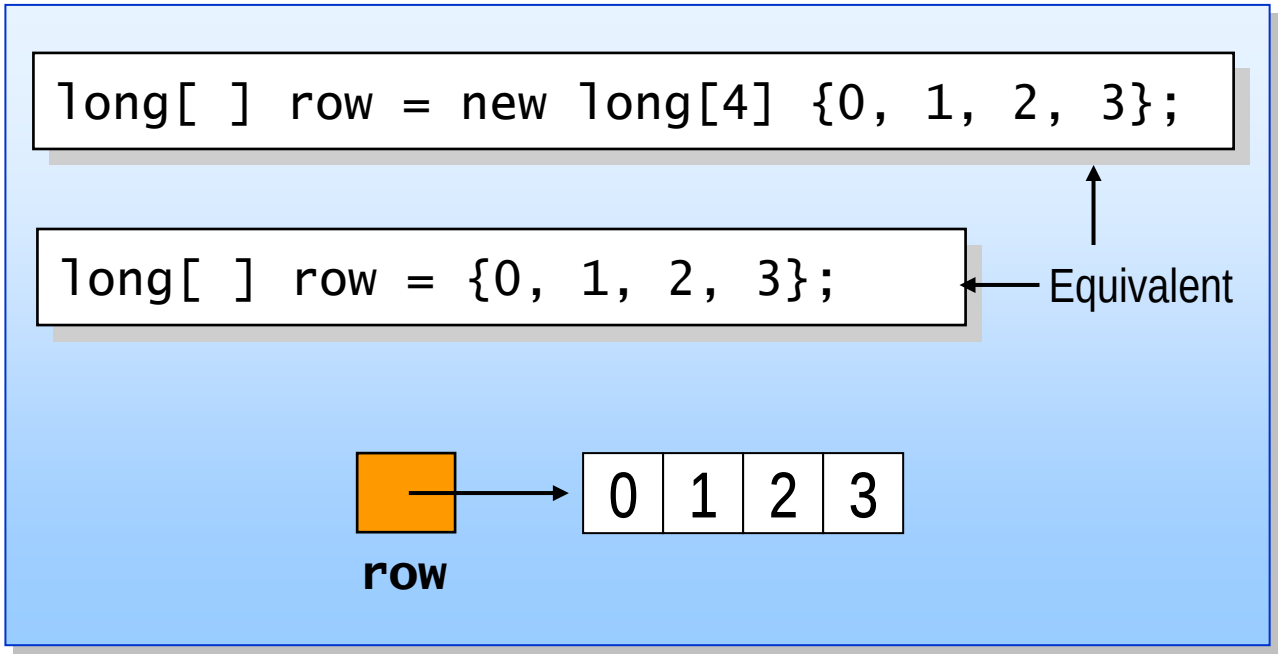
```
long[ ] row = new long[4];
```

Array size specified by run-time integer value:

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

Initializing Array Elements

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



Initializing Array Elements

Example

- int [ ] table1;
- char [ ] table2;
- float [ ] table3;
- string [ ] tableStr;

- int [ ] table1 = {17,-9,4,3,57};
- char [ ] table2 = {'a','j','k','m','z'};
- float [ ] table3 = {-15.7f, 75, -22.03f, 3 ,57 };
- string [ ] tableStr = {"chat","chien","souris","rat","vache"};

If you want to initialize and array and populate it when you declare it, C# supports a convenient syntax called the initializer list…

Understanding Default Values

- Assume you have created a **`Button`** class.
- Declare an array of Button objects with the following statement:

```
Button[] myButtonArray;
```

- and instantiate the actual array like this:

```
myButtonArray = new Button[3];
```

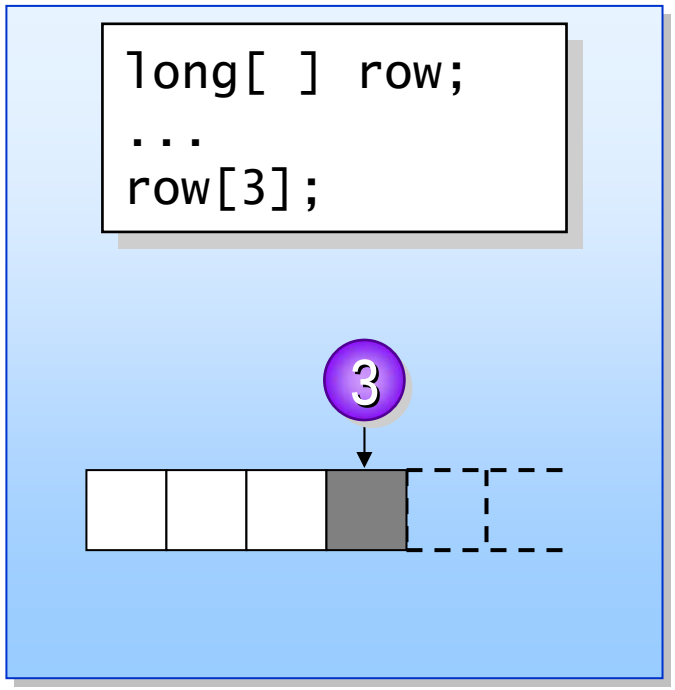
- You can shorten this to:

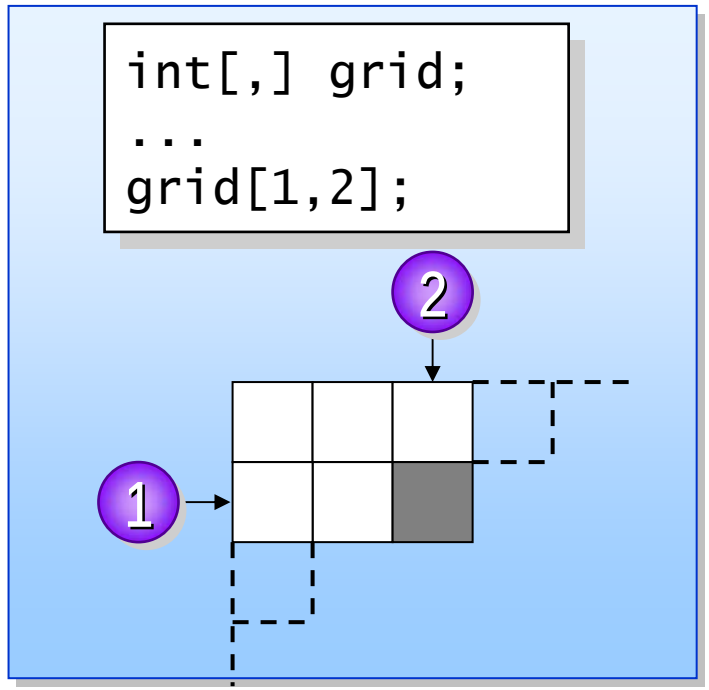
```
Button[ ] myButtonArray = new Button[3];
```

- This statement does not create an array with references to three Button objects.
  - Instead, this creates the array myButtonArray with three null references.
  - To use this array, you must first construct and assign the Button objects for each reference in the array.
  - You can construct the objects in a loop that adds them one by one to the array.

Accessing Array Elements -1

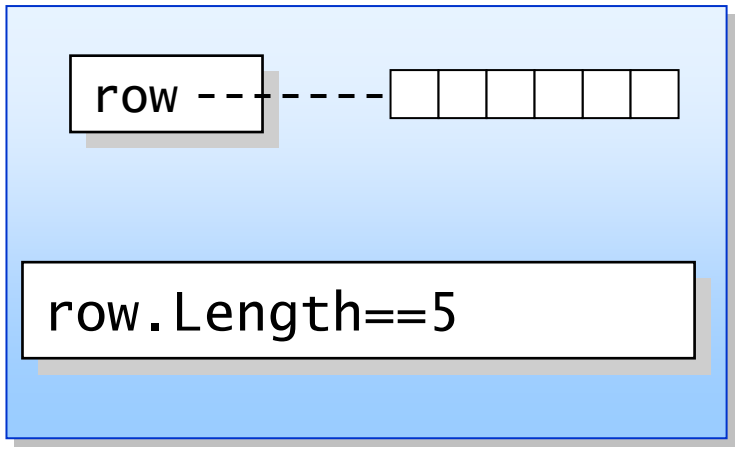
- Access the elements of an array using the index operator [ ].
- Arrays are zero-based, which means that the index of the first element is always.





Accessing Array Elements -2

- Aarrays are objects and thus have properties.
- One of the more useful of these is **`Length`**, which tells you how many objects are in an array.
- Array objects can be indexed from 0 to Length-1.
- That is, if there are five elements in an array, their indices are 0,1,2,3,4.



Accessing Array Elements -3

```
int [ ] table1 = new int [5];
table1[0] = -458;
table1[4] = 5891;
table1[5] = 72;  // Error
for (int i = 0 ; i<= table1.Length-1; i++)
  table1[i] = 3*i-1;
```

Example -1

```
public class Employee
   {
      public Employee(int empID)
      {
         this.empID = empID;
      }
      public override string ToString( )
      {
         return empID.ToString( );
      }
      private int empID;
   }
```

```
public class Tester{
   static void Main( ) {
      int[ ] intArray;
      Employee [ ] empArray;
      intArray = new int[5];
      empArray = new Employee[3];
      for (int i = 0; i< empArray.Length; i++) {
        empArray[i] = new Employee(i+5);
      }
      for (int i = 0; i<intArray.Length; i++) {
       Console.WriteLine (intArray[i].ToString( ));
      }
      for (int i = 0; i<empArray.Length;i++){
       Console.WriteLine(empArray[i].ToString( ));
      }
  }
```

Output:

```
0
0
0
0
0
5
6
7
```

foreach

- C# introduces a new keyword that helps us iterate (visit each element) of an array
- **foreach** allows you to iterate over all of the elements of an array.

```
foreach (type variable in array)
{
   // body
}
```

Visits each element of the array starting with the 0 and incrementing the index visited on each pass of the loop. With each pass through the loop, the variable takes on the value of the array at the current index.

Comparison

```
for (int i = 0; i<intArray.Length; i++) {
  Console.WriteLine (intArray[i].ToString( ));
 }
for (int i = 0; i<empArray.Length;i++){
       Console.WriteLine(empArray[i].ToString( ));
   }
```

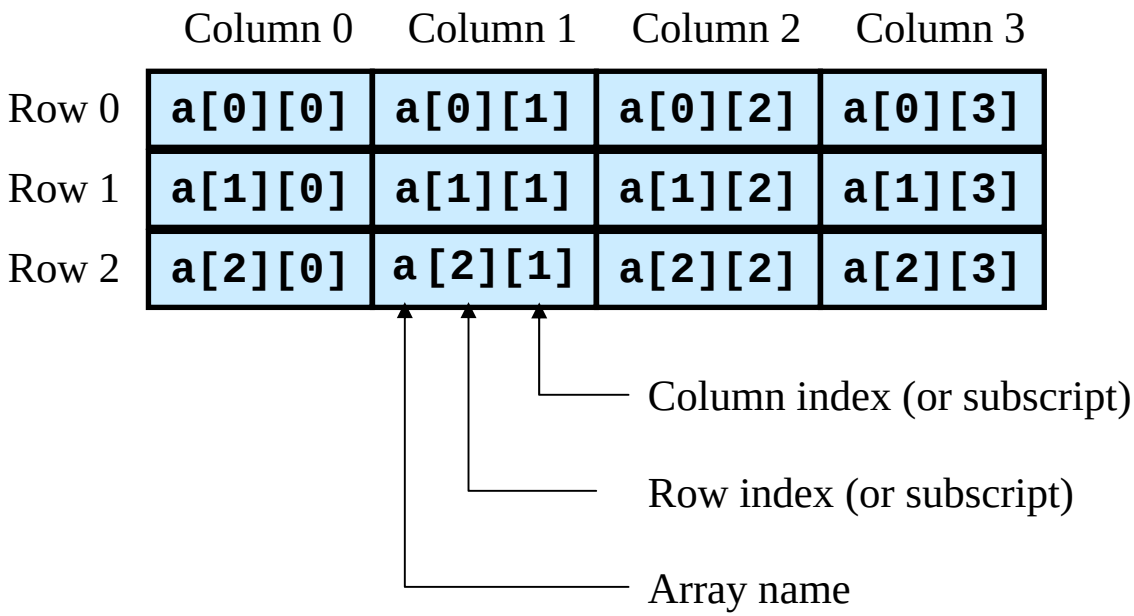
```
foreach (int i in intArray){
   Console.WriteLine(i.ToString( ));
}
foreach (Employee e in empArray){
   Console.WriteLine(e.ToString( ));
 }
```

Multidimensional Arrays

- Arrays have dimensions
  - Int [,] defines a 2-dimensional array
- C# supports 2 kinds of multidimensional arrays
  - Rectangular
  - Jagged

Rectangular arrays

- A rectangular array is an array of two (or more) dimensions.
- In the classic two-dimensional array, the first dimension is the number of rows and the second dimension is the number of columns.
- In a rectangular array, every row is the same length.



Rectangular arrays

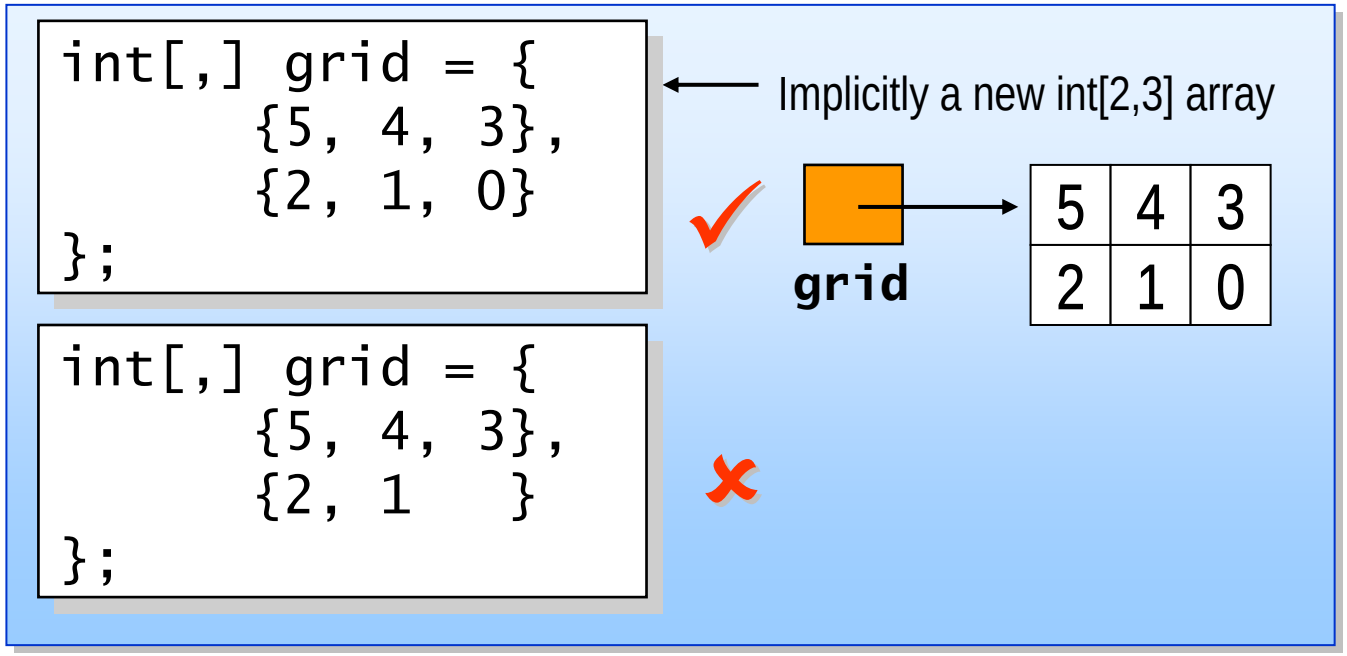
```
int [ , ] table1;
char [ , ] table2;
float [ , ] table3;
...
string [ , ] tableStr;
```

```
int [ , ] table1 = new int [5, 2];
char [ , ] table2 = new char [9,4];
float [ , ] table3 = new float [2,8];
...
string [ , ] tableStr = new String [3,9];
```

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

Initializing Multidimensional Array Elements

- You can also initialize multidimensional array elements
  - All elements must be specified



grid.Length = 2 x 3 = 6

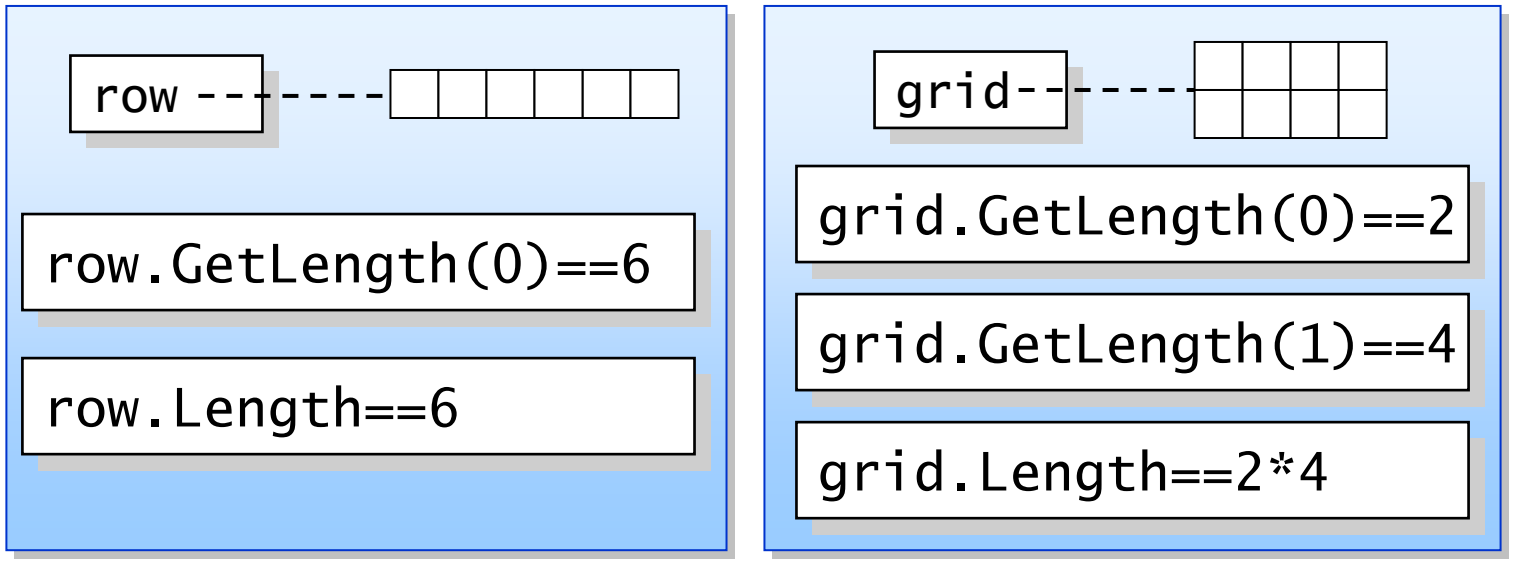
Rectangular arrays

- Useful properties and methods
  - Length: total number of elements
  - Rank: total number of dimensions
  - GetLength (int dimension): number of elements in the specified dimension
- To iterate through a rectangular array

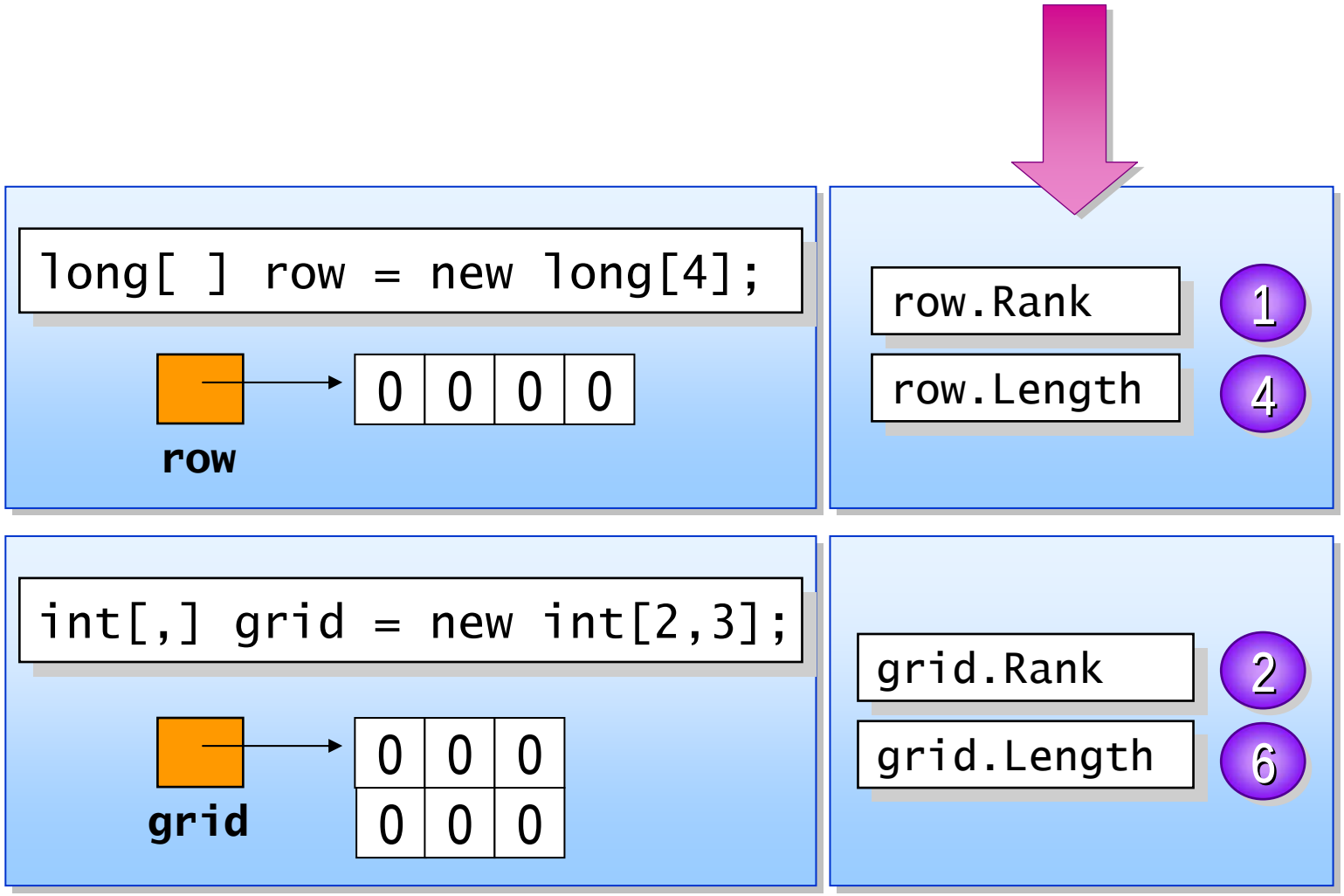
```
for(int i=0; i<a.GetLength(0); i++)
  for(int j=0; j<a.GetLength(1); j++)
  { /*Work with a[i,j]*/ }
```

Rectangular arrays : Bounds

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



Rectangular arrays : Bounds



```
public class Tester {
   static void Main( ) {
   const int rows = 4;  const int columns = 3;
   int[,] rectangularArray = new int[rows, columns];
   for (int i = 0;i < rows;i++)  {
     for (int j = 0;j<columns;j++) {
        rectangularArray[i,j] = i+j; }
    }
   for (int i = 0;i < rows;i++)  {
     for (int j = 0;j<columns;j++) {
           Console.WriteLine("rectArray[{0},{1}] =
           {2}", i,j,rectangularArray[i,j]); }
        }
     }
   }
```

- Output:
- rectArray[0,0] = 0
- rectArray[0,1] = 1
- rectArray[0,2] = 2
- rectArray[1,0] = 1
- rectArray[1,1] = 2
- rectArray[1,2] = 3
- rectArray[2,0] = 2
- rectArray[2,1] = 3
- rectArray[2,2] = 4
- rectArray[3,0] = 3
- rectArray[3,1] = 4
- rectArray[3,2] = 5

foreach for Rectangular arrays

```
string[,] a = new string[10,20];
/* code to initialize a */
...
// Iterate through a
foreach (string s in a){
    /* work with s */
}
```

Jagged arrays

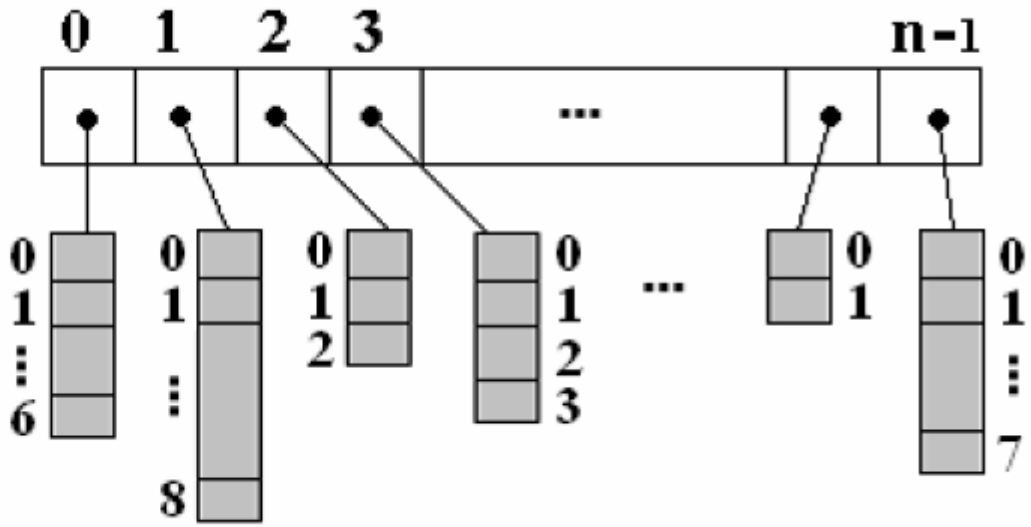
- In a 2D rectangular array all rows have the same number of columns.
- A jagged array, however, is an array of arrays, each of which can be a different length.
- When you create a jagged array, you declare the number of rows in your array.
- Each row will hold an array, which can be of any length.
- These arrays must each be declared.
- You can then fill in the values for the elements in these "inner" arrays.

Jagged arrays

- You would declare a two-dimensional jagged array of integers named myJaggedArray as follows:

```
int [ ] [ ] myJaggedArray;
```

```
int n = 10;
int [ ][ ] myArray = new int [n][ ];
myArray[0] =  new int[7];
myArray[1] =  new int[9];
myArray[2] =  new int[3];
myArray[3] =  new int[4];
…
myArray[n-2] =  new int[2];
myArray[n-1] =  new int[8];
```



Jagged arrays

```
int[][] array2 = new int[ 3 ][];
array2[ 0 ] = new int[] { 1, 2 };
array2[ 1 ] = new int[] { 3 };
array2[ 2 ] = new int[] { 4, 5, 6 };
```

- `int[][] c = new int [2][]{new int[]{1,2,3},new int []{4,5}};`
- `int[][] c = {new int[]{1,2,3},new int[]{4,5}};`

```
for (int i = 0; i < array2.Length; i++ ){
  for (int j = 0; j < array2[ i ].Length; j++ )
    Console.WriteLine({0}array2[ i ][ j ];
```

foreach for Jagged arrays

```
string[ ][ ] a = new string [2][];
a[0] = new string[5];
a[1] = new string[3];
/* code to initialize a */
...
// Iterate through a (=array of arrays)
foreach(string[] arr in a)
  foreach (string s in arr){
     /* work with s */
}
```

The `params` Keyword

- You can create a method that displays any number of integers to the console by passing in an array of integers and then iterating over the array with a foreach loop.
- The `params` keyword allows you to pass in a variable number of parameters without necessarily explicitly creating the array.

```
static long AddList(params long[ ] v)
{
    long total, i;
    for (i = 0, total = 0; i < v.Length; i++)
        total += v[i];
    return total;
}
static void Main( )
{
    long x = AddList(63,21,84);
}
```

The `params` Keyword

- Here are some additional rules:
  - A method can have only one `params` argument
  - It must be the last argument in the list
  - It can be overloaded

    More specific versions take precedence over more general arguments like object

```
public class Tester {
  static void Main( ){
     Tester t = new Tester( );
     t.DisplayVals(5,6,7,8);
     int [] explicitArray = new int[5] {1,2,3,4,5};
     t.DisplayVals(explicitArray);
    }
  public void DisplayVals(params int[] intVals){
    foreach (int i in intVals){
       Console.WriteLine("DisplayVals {0}",i);
     }
  }
```

Output:

DisplayVals 5
DisplayVals 6
DisplayVals 7
DisplayVals 8
DisplayVals 1
DisplayVals 2
DisplayVals 3
DisplayVals 4
DisplayVals 5

Array Conversions

- Conversion is possible between arrays if
  - Their dimensions are equal
  - A conversion is possible between their stored types
- An implicit conversion can occur if the elements can be implicitly converted
- An explicit conversion can occur if the elements can be explicitly converted
- If an array contains reference objects, a conversion is possible to an array of base elements

Array Conversions Example

```
namespace Programming_CSharp {
using System;
public class Employee {
  public Employee(int empID) {
     this.empID = empID;
  }
  public override  string ToString( ){
     return empID.ToString( );
  }
  private int empID;
 }
```

Array Conversions Example

```
public class Tester
 {
   // the conversion is implicit since both Employee
   // and string derive (ultimately) from object
   public static void PrintArray( object[] theArray){
      Console.WriteLine("Contents of the Array {0}",
      theArray.ToString( ));
      foreach (object obj in theArray)
        {
          Console.WriteLine("Value: {0}", obj);
         }
   }
```

Array Conversions Example

```
static void Main( ){
   Employee[] myEmployeeArray = new Employee[3];
   for (int i = 0;i < 3;i++){
        myEmployeeArray[i] = new Employee(i+5);
   }
   PrintArray(myEmployeeArray);

   string[] array = {"hello", "world"};
   PrintArray(array);
  }
 }
}
```

Array Conversions Example

```
Output:
Contents of the Array Programming_CSharp.Employee[]
Value: 5
Value: 6
Value: 7
Contents of the Array System.String[]
Value: hello
Value: world
```

System.Array

- The Array class has a number of useful methods that extend the capabilities of arrays and make them smarter than arrays seen in other languages
- Two useful static methods of Array are **`Sort( )`** and **`Reverse( ).`**
- These are fully supported for the built-in C# types such as string.
- Making them work with your own classes is a bit trickier.

```
// Reverse the order of the elements
// of a 1D array a
Array.Reverse(a);
```

```
namespace Programming_CSharp {
 using System;
  public class Tester {
    public static void PrintMyArray(object[] theArray){
        foreach (object obj in theArray){
            Console.WriteLine("Value: {0}", obj);
        }
        Console.WriteLine("\n");
    }
```

```
static void Main( ){
    String[] myArray = {"Who", "is", "John", "Galt"};
    PrintMyArray(myArray);
    Array.Reverse(myArray);
    PrintMyArray(myArray);


    String[] myOtherArray = {"We", "Hold", "These",
     "Truths","To", "Be", "Self", "Evident"};
    PrintMyArray(myOtherArray);
    Array.Sort(myOtherArray);
    PrintMyArray(myOtherArray);
    }
 }
}
```

Value: Who
Value: is
Value: John
Value: Galt

Value: Galt
Value: John
Value: is
Value: Who

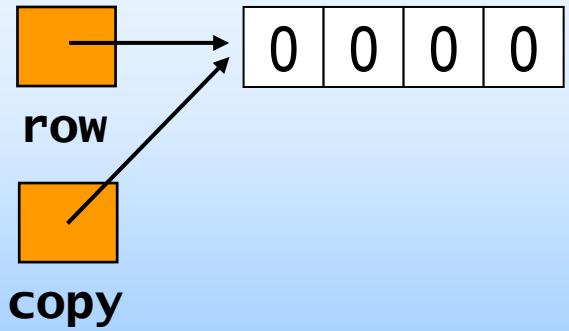
Value: We
Value: Hold
Value: These
Value: Truths
Value: To
Value: Be
Value: Self
Value: Evident

Value: Be
Value: Evident
Value: Hold
Value: Self
Value: These
Value: To
Value: Truths
Value: We

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

```
long[ ] row = new long[4];
long[ ] copy = row;
...
row[0]++;
long value = copy[0];
Console.WriteLine(value);
```



Returning Arrays from Methods

- You 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

```
class MyCopy
{
   public static void Main( )
     {
                    int[] data1 = {1, 2, 3, 4, 5, 6, 7};
                    int[] data2 = {8, 9, 10, 11, 12, 13, 14};
                    CopyIt(data1, data2);
                    Console.Write("data1:");
                    for (int i = 0; i < data1.Length; ++i)
                      Console.Write(" " + data1[i]);
                    Console.WriteLine();     // Output data1
                    Console.Write("data2:");
                    for (int i = 0; i < data2.Length; ++i)
                       Console.Write(" " + data2[i]);
      }
     static void CopyIt(int[] from, int[] to) {
                    for (int i = 0; i < from.Length; ++i)
                              to[i] = from[i];
          }
}
```

```
class SystemCopy
{
        public static void Main( )
        {
                int[] data1 = {1, 2, 3, 4, 5, 6, 7};
                int[] data2 = {8, 9, 10, 11, 12, 13, 14};
                System.Array.Copy (data1, data2, data1.Length);
                Console.Write("data1:");
                for (int i = 0; i < data1.Length; ++i)
                   Console.Write("  " + data1[i]);
                Console.WriteLine();     // Output data12
                Console.Write("data2:");
                for (int i = 0; i < data2.Length; ++i)
                      Console.Write("  " + data2[i]);
                Console.WriteLine();     // Output data2
        }
}
```

Passing Arrays by Value and by Reference

- Variables that “store” object, actually store references to those objects
- A reference is a location in computer’s memory where the object itself is stored
- Passing value types to methods
  - A copy of the variable is made
  - Any changes to variable in method do not effect the original variable
- Passing reference types to methods
  - A copy of the reference to the object is made
  - Any changes to the reference in the method do not effect the original variable
  - Any changes to the contents of the object in the method, do effect the object outside the method

Passing Arrays by Value and by Reference

- Keyword **`ref`** may be used to pass arguments to method by reference
  - Value type variables are not copied – modifying the variable in the method will modify the variable outside the method
  - References to objects are not copied – modifying the reference in the method will modify the reference outside the method
- Programmers have to be careful when using **`ref`**
  - May lead to references being set to null
  - May lead to methods modifying variable values and references in ways that are not desired

Example 1

```
static void Main()  {
    int[] firstArray = { 1, 2, 3 };
    int[] firstArrayCopy = firstArray;
    for ( int i = 0; i < firstArray.Length; i++ )
          Console.WriteLine ( firstArray[i]);
    FirstDouble(firstArray );
    Console.WriteLine (" After func :");
    for ( int i = 0; i < firstArray.Length; i++ )
          Console.WriteLine ( firstArray[ i ] );
    if ( firstArray == firstArrayCopy )
      Console.WriteLine("Its refer to same array\n");
    else
      Console.WriteLine("Its refer to different array\n");
}
```

Example 1

```
static void FirstDouble(int[] array )
{
    for ( int i = 0; i < array.Length; i++ )
          array[ i ] *= 2;
}
```

Example 2

```
static void Main() {
    int[] firstArray = { 1, 2, 3 };
    int[] firstArrayCopy = firstArray;
    for ( int i = 0; i < firstArray.Length; i++ )
       Console.Writeline({0}, firstArray[ i ] );
    FirstDouble( firstArray );
    for ( int i = 0; i < firstArray.Length; i++ )
       Console.Writeline({0}, firstArray[ i ] );

    if ( firstArray == firstArrayCopy )
     Console.Writeline(“Its refer to the same array\n“);
   else
    Console.Writelen("\nIts refer to different array\n“);
}
```

```
1 2 3
```

```
4 5 6
```

```
Its refer to the same array
```

Example 2

```
void FirstDouble( int[] array ) {
  for ( int i = 0; i < array.Length; i++ )
    array[ i ] *= 2;
    // create new reference and assign it to array
    array = new int[] { 11, 12, 13 };
}
```

Example 3

```
static void Main() {
    int[] firstArray = { 1, 2, 3 };
    int[] firstArrayCopy = firstArray;
    for ( int i = 0; i < firstArray.Length; i++ )
        Console.WriteLine({0}, firstArray[ i ] );
    FirstDouble(ref firstArray );
    for ( int i = 0; i < firstArray.Length; i++ )
       Console.WriteLine({0}, firstArray[ i ] );

    if ( firstArray == firstArrayCopy )
     Console.WriteLine(“Its refer to the same array\n“);
   else
    Console.WriteLine("\nIts refer to different array\n“);
}
```

```
1 2 3
```

```
11 12 13
```

```
Its refer to different array
```

Example 3

```
void FirstDouble( ref int[] array ) {
  for ( int i = 0; i < array.Length; i++ )
    array[ i ] *= 2;
    // create new reference and assign it to array
    array = new int[] { 11, 12, 13 };
}
```

Command-Line Arguments

- The runtime passes command line arguments to Main
  - Main can take an array of strings as a parameter
  - The name of the program is not a member of the array

```
class Example3 {
    static void Main(string[ ] args) {
        for (int i = 0; i < args.Length; i++) {
            System.Console.WriteLine(args[i]);
        }
    }
}
```

Using Arrays with foreach

- The foreach statement abstracts away many details of array handling

```
class Example4 {
    static void Main(string[ ] args) {
        foreach (string arg in args) {
            System.Console.WriteLine(arg);
        }
    }
}
```