برنامه نویسی پیشرفته
C#

۱۶ مهر ۹۸

ملکی مجد

مباحث

- Declare array variables
- Populate an array with a set of data items
- Access the data items held in an array
- Iterate through the data items in an array
- Passing arrays as parameters and return values for a method
- Copying arrays
- Method overloading
- Declaring a *params* array
- Using *params object[ ]*

Declare array variables

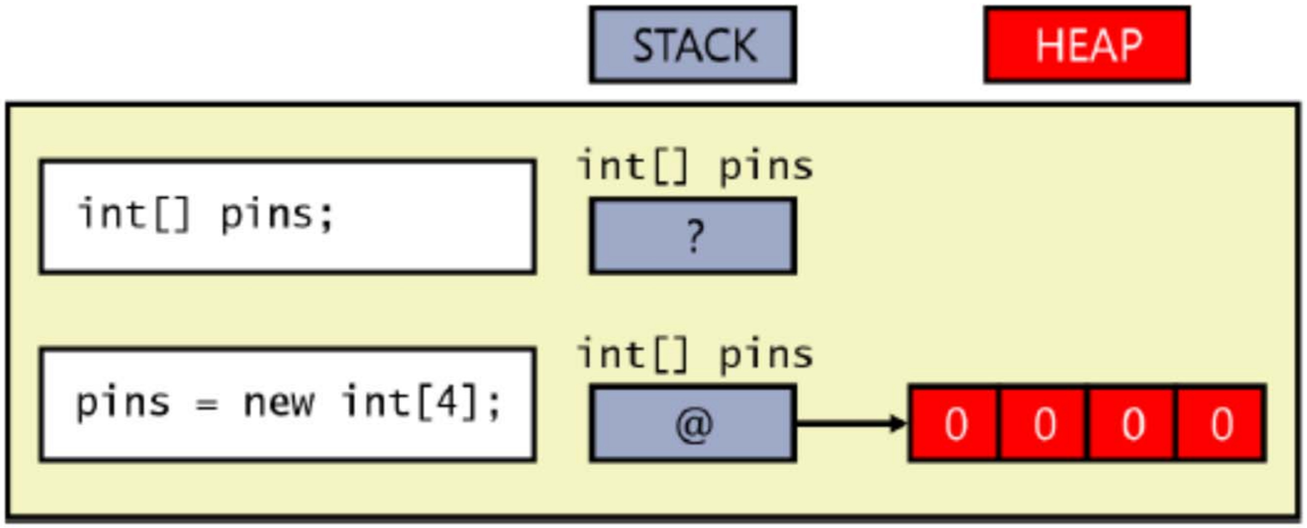
- All the items in an array have the same type
- The items in an array live in a contiguous block of memory and are accessed by using an index

- int[] pins;
- Circle[] c;

- Stack and Heap : remember the difference?
- Array items: allocated on the heap!

Declare array variables(2)

- When is memory allocated?
  - New
- the memory for the array instance is allocated dynamically
- pins = new int[4];



Populate an array with a set of data items.

- int[] pins = new int[4]{ 9, 3, 7, 2 };

- Random r = new Random();
- int[] pins = new int[4]{ r.Next() % 10, r.Next() % 10,
  r.Next() % 10, r.Next() % 10 };
- int[] pins = { 9, 3, 7, 2 };
- Time[] schedule = { new Time(12,30), new Time(5,30) };

The *System.Random* class is a pseudorandom number generator

The *Next* method returns a nonnegative random integer in the range *0* to *Int32.MaxValue* by default

Populate an array with a set of data items(2).

- Creating an implicitly typed array
  - var names = new[]{"John", "Diana", "James", "Francesca"};
  - C# compiler determines that the *names* variable is an array of strings
  - ensure that all the initializers have the same type

  - var bad = new[]{"John", "Diana", 99, 100}; error
  - var numbers = new[]{1, 2, 3.5, 99.999}; convert all to double
  - best to avoid mixing types

Access the data items held in an array

- int myPin;
- myPin = pins[2];

- myPin = 1645;
- pins[2] = myPin;

- *IndexOutOfRangeException* exception
  - specify an index that is less than 0 or greater than or equal to the length of the array

Iterate through the data items in an array

- All arrays are actually instances of the *System.Array* class in the Microsoft .NET Framework
  - For example, you can query the *Length* property to discover how many elements an array contains and iterate through all the elements of an array by using a *for* statement.

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

Iterate through the data items in an array(foreach)

```
int[] pins = { 9, 3, 7, 2 };
for (int index = 0; index < pins.Length; index++)
{
    int pin = pins[index];
    Console.WriteLine(pin);
}
```

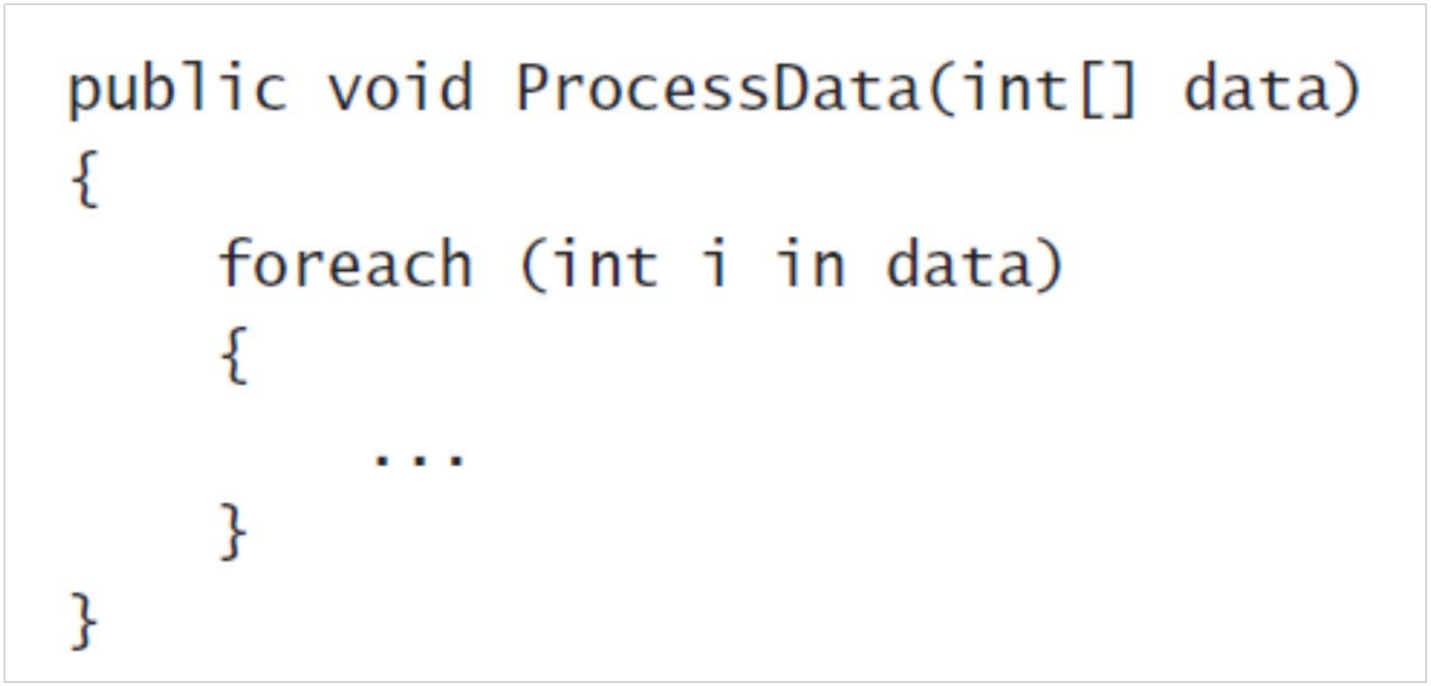
```
int[] pins = { 9, 3, 7, 2 };
foreach (int pin in pins)
{
    Console.WriteLine(pin);
}
```

Iterate through the data items in an array(foreach2)

- A *foreach* statement always iterates through the entire array. If you want to iterate through only a known portion of an array (for example, the first half) or bypass certain elements (for example, every third element), it’s easier to use a *for* statement.
- A *foreach* statement always iterates from index 0 through index *Length* – 1. If you want to iterate backward or in some other sequence, it’s easier to use a *for* statement.
- If the body of the loop needs to know the index of the element rather than just the value of the element, you have to use a *for* statement.
- If you need to modify the elements of the array, you have to use a *for* statement. This is because the iteration variable of the *foreach* statement is a read-only copy of each element of the array.

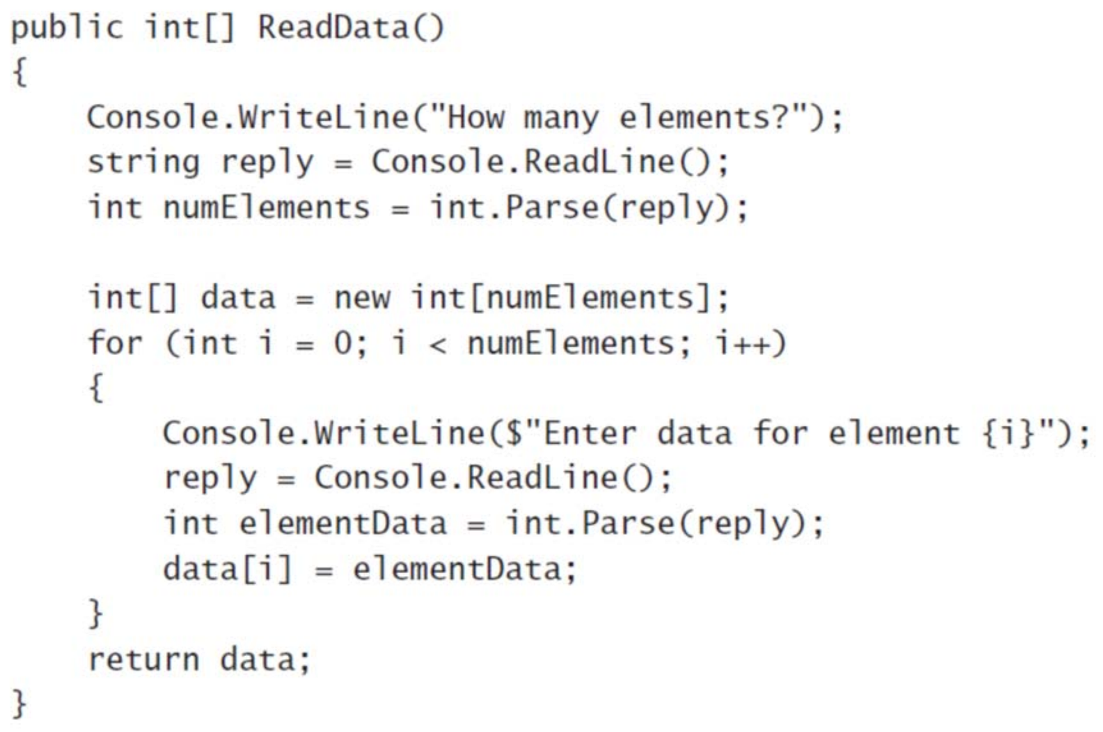
Passing arrays as parameters and return values for a method

- It is important to remember that arrays are reference objects



```
public void ProcessData(int[] data)
{
    foreach (int i in data)
    {
        ...
    }
}
```

Passing arrays as parameters and return values for a method



```
public int[] ReadData()
{
    Console.WriteLine("How many elements?");
    string reply = Console.ReadLine();
    int numElements = int.Parse(reply);

    int[] data = new int[numElements];
    for (int i = 0; i < numElements; i++)
    {
        Console.WriteLine($"Enter data for element {i}");
        reply = Console.ReadLine();
        int elementData = int.Parse(reply);
        data[i] = elementData;
    }
    return data;
}
```

Passing arrays as parameters and return values for a method

You can call the *ReadData* method like this:

```
int[] data = ReadData();
```

```
public int[] ReadData()
{
    Console.WriteLine("How many elements?");
    string reply = Console.ReadLine();
    int numElements = int.Parse(reply);

    int[] data = new int[numElements];
    for (int i = 0; i < numElements; i++)
    {
        Console.WriteLine($"Enter data for element {i}");
        reply = Console.ReadLine();
        int elementData = int.Parse(reply);
        data[i] = elementData;
    }
    return data;
}
```

Copying arrays

- An array variable contains a reference to an array instance

```
int[] pins = { 9, 3, 7, 2 };
int[] alias = pins; // alias and pins refer to the same array instance
```

if you modify the value at *pins[1]*, the change will also be visible by reading *alias[1]*.

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
 copy[i] = pins[i];
}
```

Copying arrays (CopyTo)

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
for (int i = 0; i < pins.Length; i++)
{
 copy[i] = pins[i];
}
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
pins.CopyTo(copy, 0);
```

```
int[] pins = { 9, 3, 7, 2 };
int[] copy = new int[pins.Length];
Array.Copy(pins, copy, copy.Length);
```

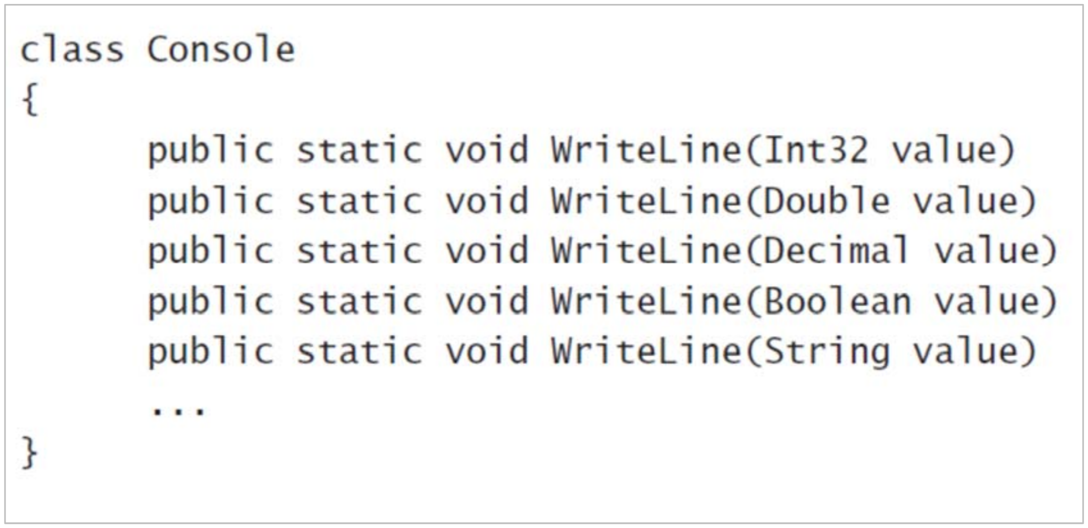
```
int[] pins = { 9, 3, 7, 2 };
int[] copy = (int[])pins.Clone();
```

the *Clone*, *CopyTo*, and *Copy* methods all create a *shallow* copy of an array

Method overloading

- *Overloading* is the technical term for declaring two or more methods with the same name in the same scope.
  - to perform the same action on arguments of different types

- Example :
  - *Console.WriteLine* method

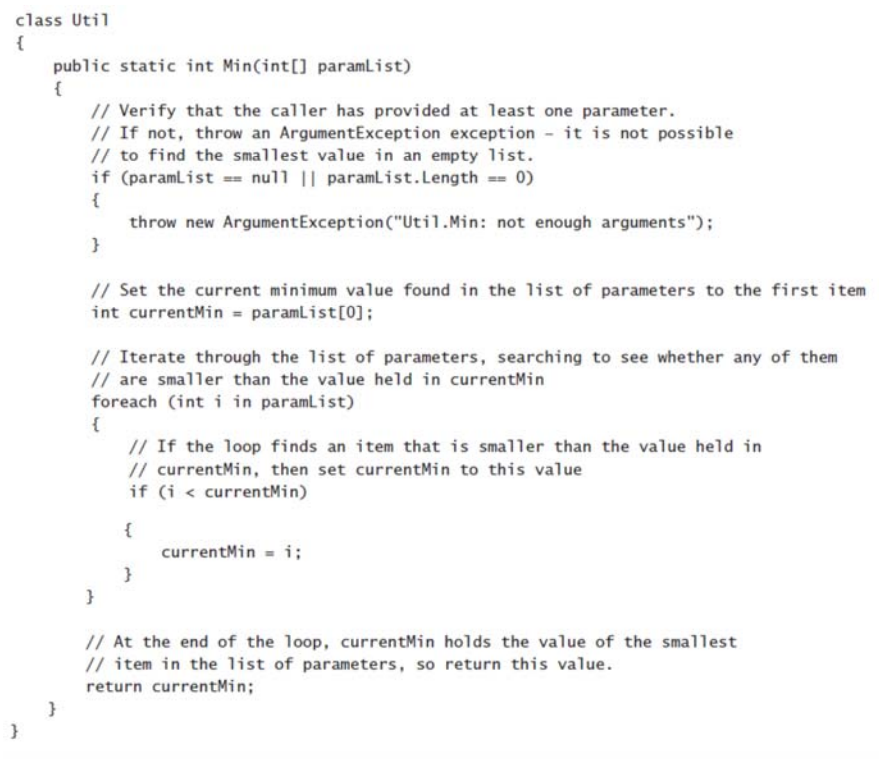


```
class Console
{
      public static void WriteLine(Int32 value)
      public static void WriteLine(Double value)
      public static void WriteLine(Decimal value)
      public static void WriteLine(Boolean value)
      public static void WriteLine(String value)
      ...
}
```

Method overloading(2)

- overloading doesn’t easily handle a situation in which the type of parameters doesn’t vary but the number of parameters does

Use array to find minimum



```
class Util
{
    public static int Min(int[] paramList)
    {
        // Verify that the caller has provided at least one parameter.
        // If not, throw an ArgumentException exception - it is not possible
        // to find the smallest value in an empty list.
        if (paramList == null || paramList.Length == 0)
        {
            throw new ArgumentException("Util.Min: not enough arguments");
        }

        // Set the current minimum value found in the list of parameters to the first item
        int currentMin = paramList[0];

        // Iterate through the list of parameters, searching to see whether any of them
        // are smaller than the value held in currentMin
        foreach (int i in paramList)
        {
            // If the loop finds an item that is smaller than the value held in
            // currentMin, then set currentMin to this value
            if (i < currentMin)

            {
                currentMin = i;
            }
        }

        // At the end of the loop, currentMin holds the value of the smallest
        // item in the list of parameters, so return this value.
        return currentMin;
    }
}
```

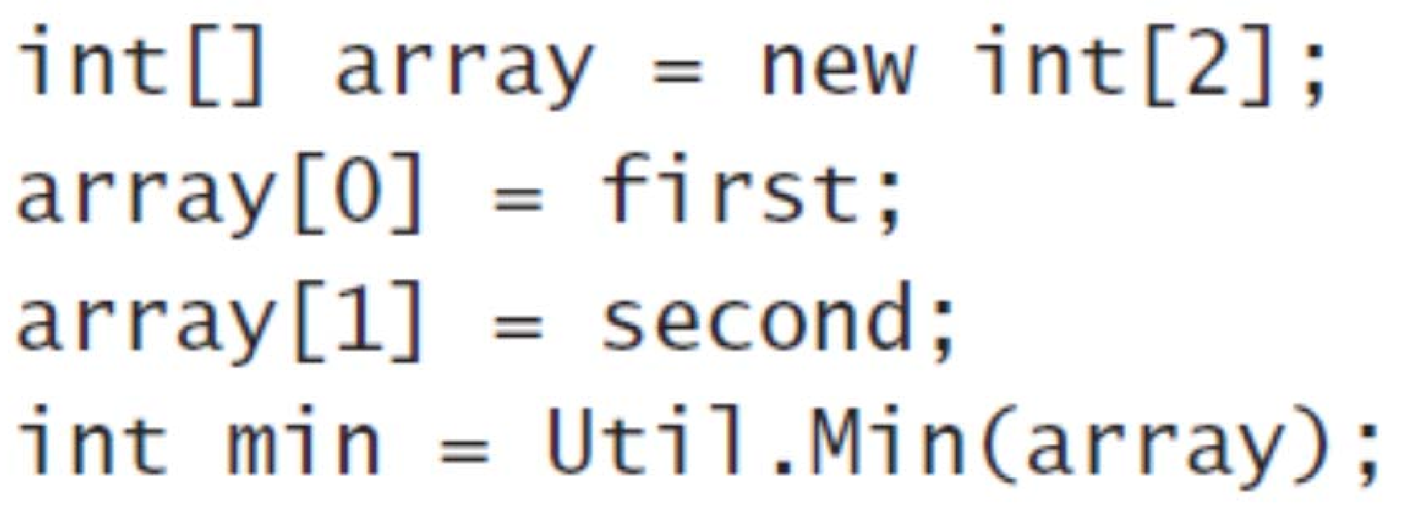
Use array to find minimum

```
 class Util
 {
     public static int Min(int[] paramList)
     {

         if (paramList == null || paramList.Length == 0)
         {
             throw new ArgumentException("Util.Min: not enough arguments");
         }
         foreach (int i in paramList)
         {
             if (i < currentMin)

             {
                 currentMin = i;
             }
         }
         return currentMin;
     }
}
```

2 integer



```
int[] array = new int[2];
array[0] = first;
array[1] = second;
int min = Util.Min(array);
```

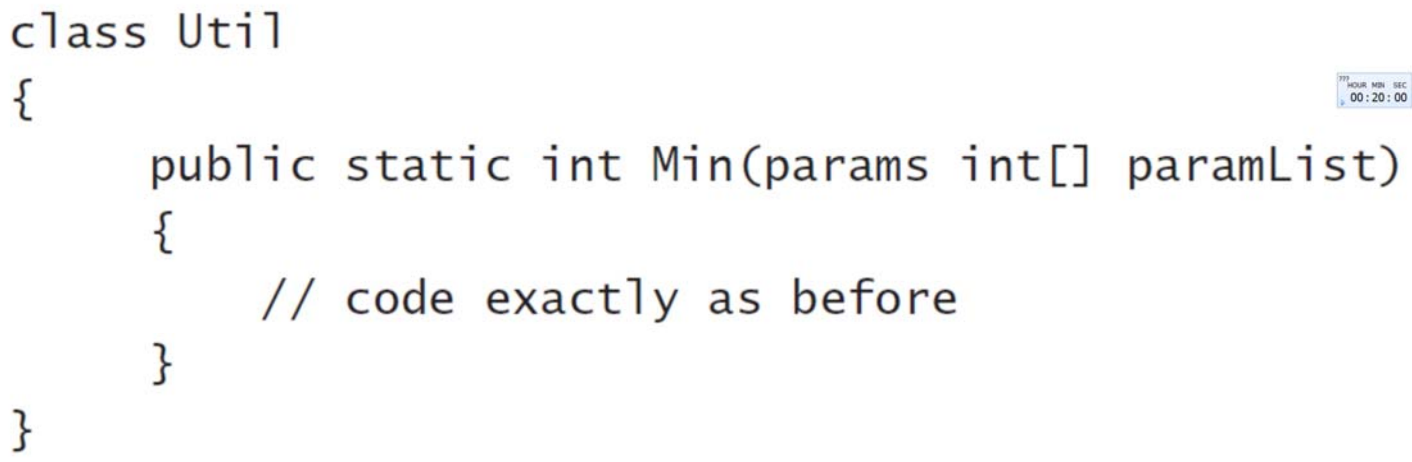
3 integer

```
int[] array = new int[3];
array[0] = first;
array[1] = second;
array[2] = third;
int min = Util.Min(array);
```

```
int min = Util.Min(new int[] {first, second, third});
```

Declaring a *params* array

- Using a *params* array, you can pass a variable number of arguments to a method
  - *params* keyword



```
class Util
{
     public static int Min(params int[] paramList)
     {
         // code exactly as before
     }
}
```

Declaring a *params* array(2)

- The effect of the *params* keyword on the *Min* method is that it allows you to call the method by using any number of integer arguments without worrying about creating an array.

- int min = Util.Min(first, second);
- int min = Util.Min(first, second, third);

- The compiler just counts the number of *int* arguments, creates an *int* array of that size, fills the array with the arguments, and then calls the method by passing the single array parameter.

params points

- You can’t use the *params* keyword with multidimensional arrays

- You can’t overload a method based solely on the *params* keyword
  - public static int Min(int[] paramList)
  - public static int Min(params int[] paramList)
- You’re not allowed to specify the *ref* or *out* modifier with *params* arrays
- A *params* array must be the last parameter
- A non-*params* method always takes priority over a *params* method
  - public static int Min(int leftHandSide, int rightHandSide)
  - public static int Min(params int[] paramList)

Using *params object[ ]*

- what if not only the number of arguments varies but also the argument type?
  - The technique is based on the facts that *object* is the root of all classes and that the compiler can generate code that converts value types (things that aren’t classes) to objects by using boxing,

- public static void Hole(params object[] paramList)
  - Black.Hole();
  - Black.Hole(null);
  - Black.Hole(new object[]{"forty two", 42});
  - Black.Hole("forty two", 42);

The *Console.WriteLine* method

- public static void WriteLine(string format, params object[] arg);

- Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}", fname,mi, lname, age);

- Console.WriteLine("Forename:{0}, Middle Initial:{1}, Last name:{2}, Age:{3}", newobject[4]{fname, mi, lname, age});