HCI 프로그래밍

3. 메소드와 인자 & 배열

HCI
Human Computer Interaction

Variable

변수 (Variable)

- 값을 임시 저장하기 위한 공간
  - 변수 값은 프로그램 수행 중 변경될 수 있음
- 변수 값은 프로그램 수행 중 변경될 수 있음
- 데이터 타입에서 정한 크기의 메모리 할당
- 반드시 변수 선언과 값을 초기화 후 사용

Variable

변수 선언과 초기화

- 자료형 (DataType)과 이름을 적어 변수를 선언

```
int radius; // 변수 선언
char c1, c2, c3; // 3 개의 변수를 한 번에 선언
double weight;
```

```
int radius = 10; // 변수 선언 및 초기화
char c1 = 'a', c2 = 'b', c3 = 'c'; // 선언과 동시에 초기값 지정
double weight = 75.56;
```

```
radius = 10 * 5; // 변수에 값 대입 (= 연산자 다음에 식)
c1 = 'r';
weight = 5.0;
```

Variable

C# 변수의 범주 (Variable categories)

- 정적 변수 (static variables)
- 인스턴스 변수 (instance variables)
- 배열 요소 (array elements)
- 값 매개 변수 (value
- 참조 매개 변수 (reference parameter
- 출력 매개 변수 (output parameters)
- 지역 변수 (local variables)

```
class A {
    public static int x; // static variable x=0
    int y; // instance variable y=0
    void Foo(int[ ] v, int a, ref int b, out int c) {
        int i = 1; // 지역변수 i는 사용하기 전에 반드시 초기화 필요함
        c = a + b++;
    }
}
```

Local Variable

- 지역변수란 메소스 내에서 선언된 변수
- 메소드가 실행될 때 변수를 저장하기 위한 메모리가 생성
- 선언된 메소드 내부에서만 사용이 가능
- 메소드의 실행이 종료될 때 메모리가 해제
- 변수를 선언한 후, 초기 값을 부여하는 초기화가 반드시 필요함

Variable

```
class MyClass {
    public static int X; // static field X=0
    public int Y; // instance field Y=0
    public void Foo()  { // instance method
        int Z; // 지역변수 Z는 선언만 된 상태
        int W = Y; // W=0
        //int U = Z; // Error: Use of unassigned local variable Z
    }
    public static void Foo2() { // static method
        X = 2;
    }
}
class Program {
    public static void Main(string[ ]  args) {
        MyClass.Foo2();
        Console.WriteLine("X = " + MyClass.X); // X=2
        //MyClass.Foo(); // Error: An object reference is required for non-static field, method, or property
    }
}
```

Class/ Method

- **C# FCL**에는 많은 클래스들이 정의되어 있음
  - Console
  - MessageBox
  - Int32
  - Math

- 클래스의 정의는 메소드**(Method)**와 자료 속성**(Data Properties)**를 포함
  - Method
    - Console.Write(), Console.WriteLine(), Int32.Parse()
  - Property
    - Int32.MinValue, Int32.MaxValue, Math.PI, Math.E

Method

- 특정 작업을 처리할 수 있도록 만들어진 코드의 묶음
- 메소드 구현
  - 여러 가지 명령문들을 조합해서 특정 작업을 처리할 수 있도록 코드를 작성하고 이름을 붙이는 것
  - 프로그램에서 메소드(함수)를 사용한다는 것은 해당 메소드(함수)가 할 수 있는 작업을 의뢰하는 것이며 "메소드(함수)를 호출한다"라고 말함
  - 클래스 내부에서만 메소드(함수)를 구현할 수 있고 호출할 수 있음

Method

메소드는 왜 필요한가?

- 문제를 작게 나누어서 해결(divide and conquer)
- 코드의 재사용
- 코드 수정의 편의성
- 검증된 코드를 사용
- 코드의 단순화와 가독성

메소드를 이용한 중복코드를 줄이는 예시

```
class NumberExample {
    public static int AbsSum(int a, int b) {
        if (a < 0) a *= -1;
        if (b < 0) b *= -1;
        return a + b;
    }
}
class Program {
    public static void Main(string[] args) {
        int a = Int32.Parse(Console.ReadLine());
        int b = Int32.Parse(Console.ReadLine());
        if (a < 0) a *= -1;
        if (b < 0) b *= -1;
        int result = a + b;
        System.out.println("결과는 " + result);
        a = Int32.Parse(Console.ReadLine());
        b = Int32.Parse(Console.ReadLine());
        if (a < 0) a *= -1;
        if (b < 0) b *= -1;
        result = a + b;
        Console.WriteLine("결과는 " + result);
        Console.WriteLine("결과는 " + NumberExample.AbsSum(-2, -3));
        }
}
```

```
-2
3
결과는 5
2
-3
결과는 5
결과는 5
```

Method

메소드 선언 예시

```
class MyClass
{
  …
  public static int  SquareSum( int num1, int num2 )
  {
   int sum = num1 + num2;
   return sum * sum;
  }
….
```

properties

Return type

Method name

parameter list

body

return value

Method 정의와 호출

- 접근 지정자
  - 메소드를 호출할 수 있는 범위를 지정함
- 반환형
  - 메소드를 돌려주는 값의 형식을 지정함
- 메소드 이름
  - 메소드를 호출할 때 사용하는 함수의 이름
- 인자 리스트
  - 메소드 실행 전에 전달되는 변수를 의미하며, 변수의 형식과 실제 값이 전달됨

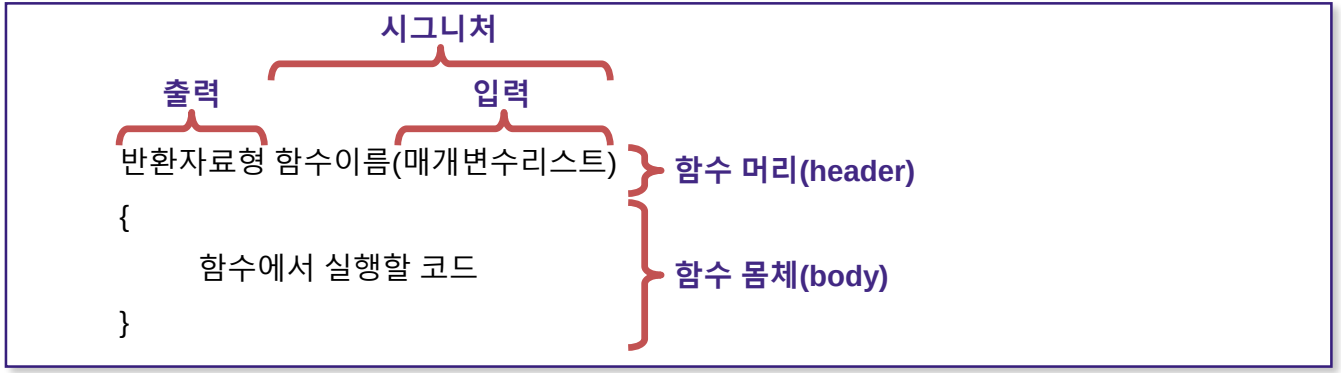
Method 정의와 호출

- 명령문
  - 실제 메소드가 실행하는 문장

- 반환 값
  - 메소드가 종료될 때 되돌려주는 값으로 위에 선언한 리턴형에 맞는 형식의 값이어야 함
  - 리턴 값은 없을 수도 있으며 이때의 리턴형은 `void`로 선언함

Method 정의

메소드(method)는 머리(header)와 몸체(body)로 구성됨



- 함수 시그니처(function signature)
  - 메소드를 구별하는데 사용됨
- 구현
  - 메소드에서 처리해야 하는 작업을 코드로 작성한 부분

Method 정의

- 메소드 이름은 식별자 이름 짓는 규칙을 따름
  - 하는 일이 무엇인지 알 수 있게 단어를 조합
  - 대문자 영문 알파벳으로 시작하고 영문자와 숫자의 조합 사용

Method 정의

메소드 매개변수 리스트

- 메소드에 입력으로 전달될 값을 저장할 변수를 정의
- 두 개 이상의 변수가 정의되면 ',' 로 분리
- 왜 매개변수를 사용할까?
  ✓매개변수를 사용하면 융통성과 재사용성을 높일 수 있음
- 메소드에 인자를 전달할 때 자동 형변환 발생
- **in, ref** 또는 **out** 없이 메서드에 대해 선언된 매개 변수는 **값**으로 호출된 메서드에 전달됨
- **params**는 이 매개 변수가 가변 개수의 인수를 사용할 수 있음을 지정함

Method 정의

반환 자료형(return type)

- 메소드에서 반환하는 결과 값의 자료형을 명시
- 메소드 내부의 코드 실행을 중단시키고 실행 흐름을 메소드 호출한 곳으로 되돌림
- 반환값 없이 메소드 실행 중단

```
return;
```

- 값(value type 또는 reference type)의
  - ✓ 값형식은 값을 리턴
  - ✓ 참조형식은 레퍼런스를 리턴

```
return 값;     // 값으로 반환
```

- **C# 7.0부터 참조로 값을 반환**
  - ✓ **ref** 키워드가 **메서드 시그니처**에 사용되고 각 **return 키워드 뒤에 오면** 값이 호출자에 참조로 반환

```
return ref 값; // 참조로 반환
```

Method 호출

```
class Program {
    static float Sum(float a, float b) {
        return a + b;
    }
    static void Main(string[ ]  args) {
        float value = Sum(3, 4); // int->float implicit type conversion
        value = Sum(3.2, 4.5); // compile error: cannot convert from double to float
        Console.WriteLine("Hello ……?");
    }
}
```

class

(parameters)

method

dot

Method 호출

static method (정적 메소드) 호출

같은 클래스 내에서의 static method 호출 :

메소드명();

다른 클래스 내에서의 static method 호출 :

클래스명.메소드명();

Method 호출

instance method (인스턴스 메소드) 호출

같은 클래스 내에서의 instance method 호출 :

메소드명();

다른 클래스 내에서의 instance method 호출 :

인스턴스명.메소드명();

Method 호출

instance vs static 메소드 호출 예제

```
namespace MethodExample {
        public class A {
                public void MethodC() { // instance method
                        Console.WriteLine("MethodC() in class A");
                }
                public static void MethodA() { // static method
                        Console.WriteLine("MethodA() in class A");
                }
        }
        public class B {
                public static void Main(string[ ]  args) {
                        A.MethodA();
                        A a = new A();
                        a.MethodC();
                }
        }
}
```

```
MethodA()  in class A
MethodC()  in class A
```

Method 호출

- 메소드 호출 시, 메소드끼리 서로를 계속 호출하여, 프로그램이 종료되지 않는 무한루프에 빠지지 않도록 주의할 것

```
using System;
namespace NestedMethodExample
{
        public class NestedMethod    {
                public static void MethodA() {
                        Console.WriteLine("MethodA.");
                }
                public static void MethodB()  {
                        MethodA();
                        Console.WriteLine("MethodB.");
                        MethodA();
                }
                public static void Main(string[ ]  args) {
                        MethodB();
                        MethodA();
                }
        }
}
```

```
MethodA.
MethodB.
MethodA.
MethodA.
```

Math Class Method

Method	Description	Example
Abs(x)	absolute value of ***x***	**Abs(23.7)** is **23.7** **Abs(-23.7)** is **23.7**
Ceiling(x)	rounds ***x*** to the smallest integer not less than ***x***	**Ceiling(9.2)** is **10.0** **Ceiling(-9.8)** is **-9.0**
Cos(x)	trigonometric cosine of ***x*** (***x*** in radians)	**Cos(0.0)** is **1.0**
Exp(x)	exponential method ***ex***	**Exp(1.0)** is approximately **2.7182818284590451** **Exp(2.0)** is approximately **7.3890560989306504**
Floor(x)	rounds ***x*** to the largest integer not greater than ***x***	**Floor(9.2)** is **9.0** **Floor(-9.8)** is **-10.0**
Log(x)	natural logarithm of ***x*** (base *e*)	**Log(2.7182818284590451)** is approximately **1.0** **Log(7.3890560989306504)** is approximately **2.0**
Max(x, y)	larger value of ***x*** and ***y*** (also has versions for **float**, **int** and **long** values)	**Max(2.3, 12.7)** is **12.7** **Max(-2.3, -12.7)** is **-2.3**
Min(x, y)	smaller value of ***x*** and ***y*** (also has versions for **float**, **int** and **long** values)	**Min(2.3, 12.7)** is **2.3** **Min(-2.3, -12.7)** is **-12.7**
Pow(x, y)	***x*** raised to power ***y*** (xy)	**Pow(2.0, 7.0)** is **128.0** **Pow(9.0, .5)** is **3.0**
Sin(x)	trigonometric sine of ***x*** (***x*** in radians)	**Sin(0.0)** is **0.0**
Sqrt(x)	square root of ***x***	**Sqrt(900.0)** is **30.0** **Sqrt(9.0)** is **3.0**
Tan(x)	trigonometric tangent of ***x*** (***x*** in radians)	**Tan(0.0)** is **0.0**

Recursive Method

재귀 메소드

- 자기 자신을 호출하는 메소드로서 같은 반복된 작업이 필요한 경우를 구현한 메소드

(예제) 재귀메소드를 이용하여 N!(factorial) 구하기

N-팩토리얼 = N * (N-1) * (N-2) * (N-3) * ……. * 1

재귀 메소스 예제

```
namespace RecursiveCallExample {
   public class RecursiveCall          {
      public static ulong Factorial(ulong number) {
         if (number <= 1)
            return 1;
         else
            return number * Factorial(number – 1);
      }

      public static void Main(string[] args) {
         ulong nfact = Factorial(5);
         Console.WriteLine("5 * 4 * 3 * 2 * 1 = " + nfact);
      }
   }
}
```

```
5 * 4 * 3 * 2 * 1
= 120
```

Params 키워드

C# params

- 메소드의 매개변수 갯수를 미리 알 수 없는 경우도 있는데, 이런 경우 C# 키워드 params를 사용함
- 이 params 키워드는 가변적인 배열을 인수로 갖게 해주는데, 파라미터들 중 반드시 하나만 존재해야 하며, 맨 마지막에 위치해야 함

```
static int Sum(int start, params int[ ]  list) {
        int sum = start;
        for (int i = 0; i < list.Length; i++) sum += list[i];
        return sum;
}
static void Main(string[ ]  args) {
        float value = Sum(10, 3, 5, 7, 9); // value = 34
}
```

Extension Method (C# 3.0)

확장 메소드란 특수한 종류의 **static** 메소드

- 다름 클래스/구조체의 인스턴스 메소드인 것처럼 사용되는 기능을 제공함
- 클래스/구조체/인터페이스 등에 적용할 수 있음
- 참조만 할 수 있는 클래스/구조체에 이미 만들어진 메서드를 응용하거나 또 다른 메서드를 추가해서 사용할 수 있는 것이 확장 메소드임
  - .NET Framework에 배포된 클래스/구조체
  - int형이나 string형에도 확장 메서드를 추가해서 사용할 수 있음

확장 메소드의 제약조건

- 확장 메소드를 가진 클래스는 **static class**이어야 함
- 확장 메소드는 **static method**이어야 함
- 확장 메소드의 **첫번째 매개변수에는 this 키워드**가 들어가야 함

Extension Method

Extension Method 예제

```
public static class MyExtensions {
    public static bool IsEvenNumber(this int num) => num % 2 == 0;
    public static int WordCount(this String str) {
        return str.Split(new char[ ]  { ' ', '.', '!' },
                        StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
public Program        {
    static void Main(string[ ]  args) {
        int num = 10;
        bool even = num.IsEvenNumber();
        Console.WriteLine(even); // True
        string str = "This is HCI Programming! ";
        int count = str.WordCount();
        Console.WriteLine(count); // 4
    }
}
```

Partial Method (C#

C#2.0	C#3.0	Partial method
partial 키워드가 도입되어, partial class, partial struct, partial interface를 사용할 수 있었음	partial method 지원함	메소드가 반드시 private 이어야 하고, 리턴값이 void 이어야 함

Patial Method

Patial Method 예제

```
public partial class Class2   {
        public void Run() {
                DoThis();
        }
        // 접근한정자가 없음 기본 private
        // void 반환
        // out 매개변수가 없음
        // 한정자 virtual, override, sealed, new, extern 없음
        partial void DoThis(); // 메소드 시그니처 정의
}
public partial class Class2   {
        partial void DoThis() {  // 메소드 구현
                Log(DateTime.Now);
        }
}
```

Named Parameter (C# 4.0)

Named parameter

- C#은 메소드에 매개변수를 전달할 때, 일반적으로 매개변수 위치에 따라 순차적으로 매개변수가 넘겨지게 되는데, C# 4.0부터 위치와 상관없이 매개변수를 지정하여 전달할 수 있게 하였음
  - 이것을 named parameter라고 부름

```
static float Sum(float a, float b) {
    return a + b;
}
static void Main(string[ ]  args) {
    float value = Sum(b: 3.2f, a: 4.5f);
}
```

Optional Parameter (C# 4.0)

Optional Parameter

- C# 4.0부터 어떤 메소드의 매개변수가 기본값을 갖고 있다면, 메소드 호출 시 이러한 매개변수를 생략하는 것을 허용함
  - 이것을 optional parameter라고 부름
- 단, optional parameter는 반드시 매개변수 중 맨 마지막에 놓여져 있어야 함
- 복수개의 optional parameter가 있는 경우 반드시 optional이 아닌 매개변수들 뒤에 위치해야 함

```
static float Sum(float a, float b=23.5f) {
    return a + b;
}
static void Main(string[] args) {
    float value = Sum(3.2f); // a=3.2, b=23.5, value=26.7
}
```

Local Function (C# 7.0)

- 로컬 함수 **(Local Function)**는 멤버 내의 중첩 메소드로 정의함. 컴파일러가 동일한 메서드로 취급
- 로컬 함수 **(Local Function)**는 해당 메소드 이외에서 호출할 수 없으며, 하나의 메소드 안에는 여러 개의 로컬 함수를 만들 수 있음.
- 모든 로컬함수는 **private** 이며, 일반 메소드 정의와 달리 로컬 함수 정의에는 멤버 액세스 한정자를 포함할 수 없음
- 로컬 함수는 다음 한정자를 사용 가능함
  - `async`
  - `unsafe`
  - `static` (C#8.0 이상) 정적 로컬 함수는 지역변수 또는 인스턴스 상태를 캡처할 수 없음
  - `extern` (C#9.0 이상) 외부 로컬 함수는 `static`이어야 함

Local Function (C# 7.0)

- 해당 메소드 매개 변수를 비롯하여 포함하는 멤버에 정의된 모든 지역 변수는 비정적 로컬함수에서 호출 가능함
  - **로컬함수는 클로져(Closure) 기능을 사용할 수 있음**

```
public static ref int Calc(int a, int b, int c) { // a=2, b=5, c=7
    int factor = 10; // Calc 내에 지역변수 factor를
    int ab = Formula(a, b); // 34%10 = 4
    int ac = Formula(a, c);  // 48%10 = 8
    return Math.Max(ab, ac); // 8
    int Formula(int x, int y) {
        int res = 2 * x + 7 * y – 5;
        return res % factor; // 로컬함수 내부에서 사용가능 - closure
    }
}
static void Main(string[ ]  args) {
    int number = Calc(2, 5, 7); // 8
}
```

Local Function (C# 7.0)

```
private static string GetText(string path, string filename) {
    var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
    var text = reader.ReadToEnd();
    return text;
    string AppendPathSeparator(string filepath) {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\");
    }
}
```

```
#nullable enable
// C#9.0부터 로컬 함수, 매개 변수, 형식 매개 변수에 특성을 적용가능
private static void Process(string?[ ]  lines, string mark) {
    foreach (var line in lines) {
        if (IsValid(line)) {
            // processing…
        }
    }
    bool IsValid([NotNullWhenTrue] string? line) {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length; // closure
    }
}
```

Array

- 배열 **(array)**이란 같은 형식의 데이터를 그룹화해서 사용하거나 편집할 때 유용하게 사용할 수 있도록 하는 데이터 타입
  - System.Array라는 클래스에서 파생되었으며 배열 (array)의 모든 element 는 형(type)이 같아야 함
  - 구조체 (structure)란 서로 연관성이 있는 데이터이지만 형식 (type)이 다른 경우의 그룹화에 사용
  - 배열 (array)는 서로 연관성이 있고 형식도 같은 데이터를 그룹화 할 때 사용함

Array

배열의 특징

- 같은 데이터형의 변수를 한꺼번에 여러 개 생성
- 배열의 크기는 배열의 첨자로 결정
- 첨자에 해당하는 만큼의 같은 데이터 형을 가진 메모리 생성
- 배열의 메모리는 연속적으로 지정
- 배열의 참조 값을 이용하여 핸들 할 수 있음
- 배열의 이름은 연속된 변수들을 참조하기 위한 참조 값
- 배열의 요소는 변수

Array

배열 선언

```
type [ ] name;
```

- type – 배열을 실제로 구성하는 요소의 형식(type)을 나타냄
- [ ] – 배열의 차원 (rank)를 나타냄
- name – 배열변수의 이름을 나타남

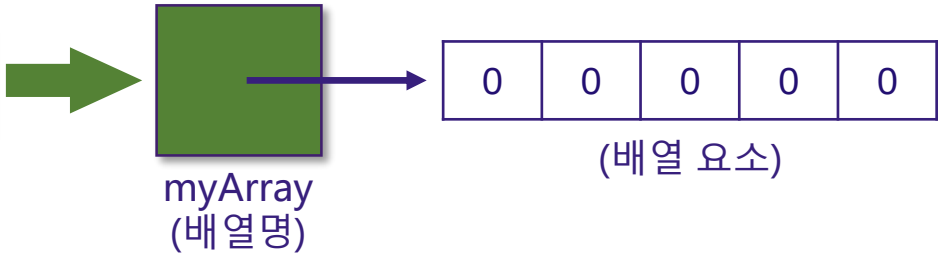
```
//선언
int[ ]  myArray;

//초기화
myArray = new int[5];
```

배열의 초기화

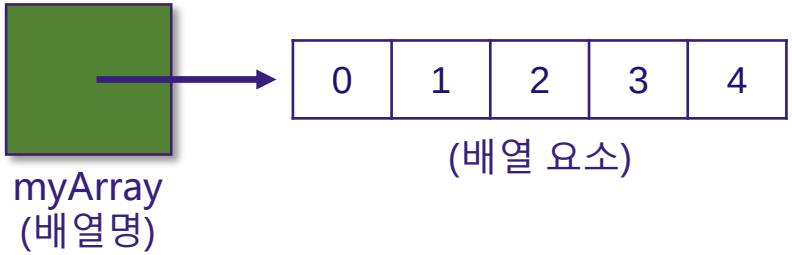
타입	초기 값
숫자(int, long, float) 등	0
문자(char)	null(빈 값을 의미)
문자열(string)	null(빈 값을 의미)
enum	0
참조형(reference)	null(빈 값을 의미)

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



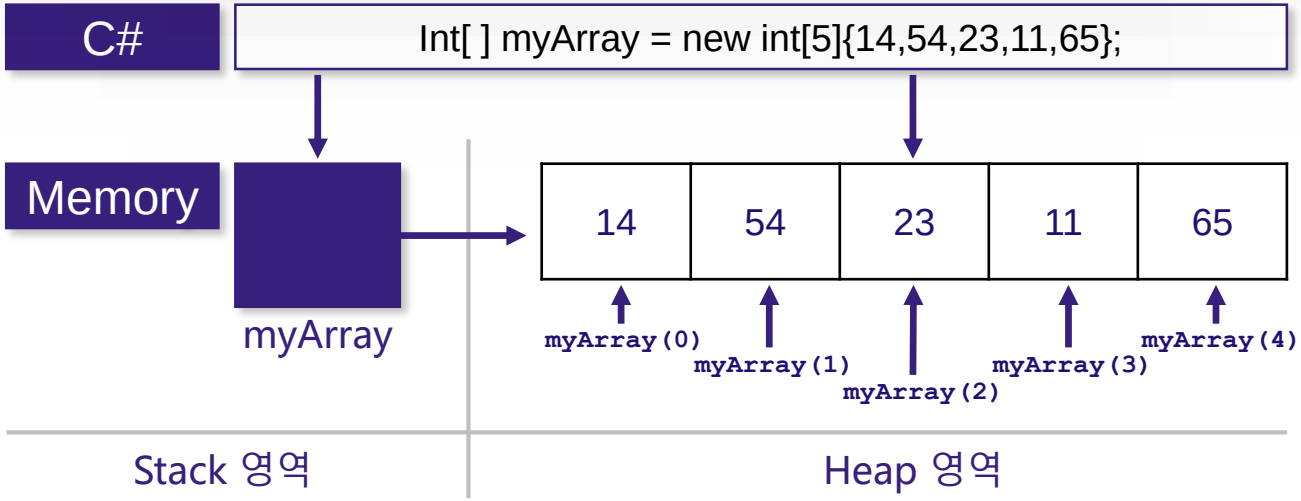
배열의 요소 값 지정하는 방법

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



배열의 초기화 후의 모습

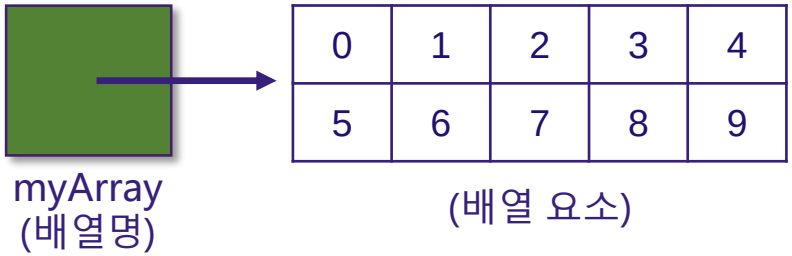
- int형의 5개의 요소를 가진 배열 myArray가 정상적으로 초기화가 되면, 메모리에 배열이 저장되는데, 배열명은 Stack에, 각 요소들의 값은 Heap에 저장이 됨



다차원 배열 선언

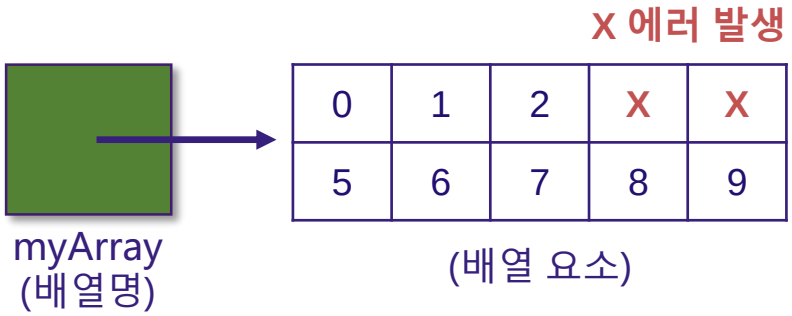
```
// 2차원 배열의 예

int[,] myArray = new int [2,5]{
        {0,1,2,3,4},
        {5,6,7,8,9}
};
```



다차원 배열 선언시의 에러 예제

```
int[,] myArray = new int[2,5]{
        {0,1,2,},
        {5,6,7,8,9}
};
```

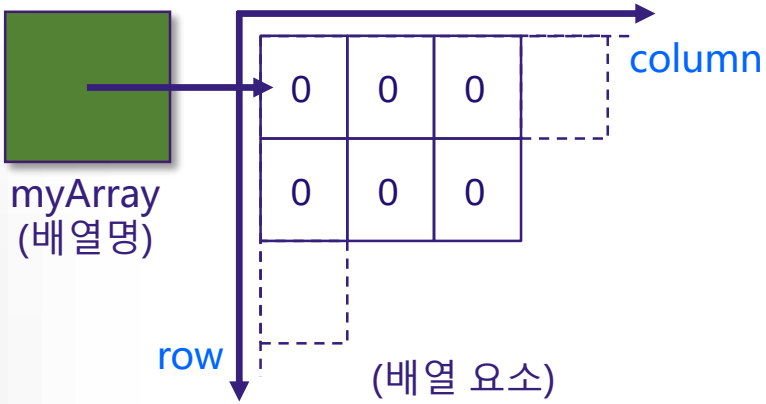


다차원 Array

2차원 배열의 크기

```
int[,] myArray = new int[rows, columns];
```

- Rows * columns 만큼의 배열 크기가 생성됨
- 실제 사용할 배열의 크기보다 훨씬 큰 배열을 선언하고 사용한다면, 메모리 낭비임
- 반대로 사용할 배열보다 선언한 배열의 크기가 작다면 배열의 크기가 가변적이지 않기 때문에 다시 다른 변수명으로 배열을 새로 선언하고 사용해야 함



불규칙적 Array

다차원 배열 vs. 불규칙적 배열

- 다차원 배열의 경우는 반드시 값을 채워줘야 함
- 불규칙적인 배열(**Jagged Array**, a.k.a **Array of Arrays**)은 [ ] [ ] 형태로 해주어야 함

```
int[,] myArray = new int[2,5]{
        {0,1,2,},          // 에러 발생
        {5,6,7,8,9}
};
```

```
int[ ] [ ] myArray = new int[2][ ] ;
myArray[0] = new int[ ]  {0,1,2,};
myArray[1] = new int[ ]  {5,6,7,8,9};
```

Array의 복사

배열의 복사

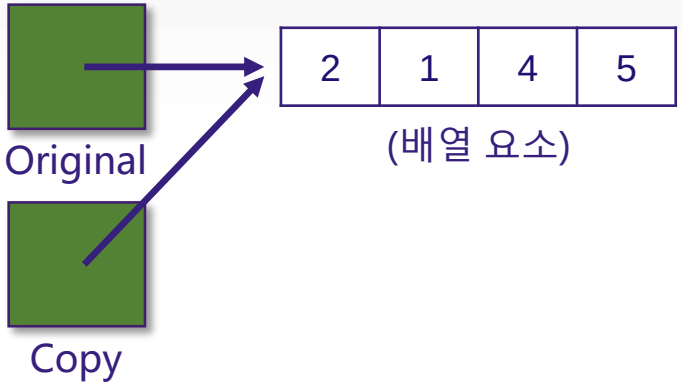
```
class CopyArray
{
        static void Main (string [ ]  args )
        {
                long [ ]  Original = new long[4] { 2, 1, 4, 5};
                long [ ]  Copy = Original;
                Console.WriteLine("1 : " + Copy[3]); // 1 : 5

                Original[3] = Original[0] + Original[2];
                Console.WriteLine("2 : " + Copy[3]); // 2 : 6
        }
};
```

Array의 복사

배열의 복사

- **배열의 복사가 이루어지면, 복사된 배열명은 새로 배열을 생성하는 것이 아니라, 원래 있는 배열의 값들을 참조하고 있는 것**
- 그렇기 때문에 원래의 배열의 요소 값에 변화가 생기면 당연히 복사된 배열에서도 바뀐 값을 참조하는 것임



배열의 차원 (rank)

```
class ArrayTest
{
        static void Main(string[ ]  args)
        {
                int[ ]  Array1 = new int[4];
                int[,] Array2 = new int[2,3];
                int[,,] Array3 = new int[2,4,3];
                 // 배열의 차원은 배열 선언시의 각 요소의 갯수
                // Array1의 차원 : 1
                Console.WriteLine("Array1의 차원 : " + Array1.Rank);
                // Array2의 차원 : 2
                Console.WriteLine("Array2의 차원 : " + Array2.Rank);
                // Array3의 차원 : 3
                Console.WriteLine("Array3의 차원 : " + Array3.Rank);
        }
}
```

Array의 활용

배열의 크기 (length)

```
class ArrayTest
{
        static void Main(string[ ]  args)
        {
                int[ ]  Array1 = new int[4];
                int[,] Array2 = new int[2,3];
                int[,,] Array3 = new int[2,4,3];
                 // 배열의 크기는 배열의 각 요소 크기의 곱셈
                // Array1의 크기 : 4
                Console.WriteLine("Array1의 크기 : " + Array1.Length);
                // Array2의 크기 : 6
                Console.WriteLine("Array2의 크기 : " + Array2.Length);
                // Array3의 크기 : 24
                Console.WriteLine("Array3의 크기 : " + Array3.Length);
        }
}
```

Array의 활용

배열의 인덱스 (index)

- **배열명[index]**라고 쓰면 그 배열의 index 순서에 있는 element를 뜻하며, 첫번째 element의 index는 0임
- 배열의 유효 index 범위를 넘는 인덱스를 사용하면 IndexOutOfRangeException 예외가 발생함

```
class ArrayTest
{
        static void Main(string[ ]  args)
        {
                int[ ]  Array1 = new int[4] {1, 2, 3, 4};
                for (int i = 0; i < Array1.Length; i++)
                {
                        Console.WriteLine("Array1[{0}]={1}", i, Array1[i]);
                }
        }
}
```

Array의 활용

- 정렬 (Sort) 메소드 – System.Array.Sort()
  - 정렬 메소드는 배열의 요소값들을 크기의 순서대로 작은 순서부터
    큰 순서대로 정렬을 해주고 이를 배열에 반영해주는 메소드
- 초기화 (Clear) 메소드 – System.Array. Clear()
  - 배열의 각 요소들의 값을 초기화하는 메소드
- 복제 (Clone) 메소드 – System.Array. Clone()
  - 배열의 크기와 요소값을 모두 같게 하여 새로운 배열을 생성하는 메소드
- 색인 (IndexOf) 메소드 – System.Array. IndexOf()
  - 찾으려는 값이 배열의 몇 번째 요소인지를 반환하는 메소드

Array의 활용

복제 (Clone), 색인 (IndexOf), 정렬 (Sort) 메소드

```
int[ ]  one = new int[ ]  {2, 1, 4, 5};          // one {2, 1, 4, 5}
int[ ]  clone = (int[ ] )one.Clone();            // clone {2, 1, 4, 5}

one[3] = one[0] + one[2];                        // one {2, 1, 4, 6} clone {2, 1, 4, 5}
foreach(int i in clone) {
        Console.Write("{0} ", i);                // 2 1 4 5
}

int where = Array.IndexOf(clone, 4);             // 4 is located in 2
Console.WriteLine("\n4 is located in {0}", where);

Array.Sort(clone);
Console.Write("After sort: ");
foreach(int i in clone) {
        Console.Write("{0} ", i);                // After sort: 1 2 4 5
}
```

Array의 활용

메소드의 리턴 값으로의 배열

```
class ArrayReturn
{
    static void Main(string[ ]  args)
    {
        int[ ]  MyArray = CreateIntArray(10);
         // MyArray의 크기 : 10
        Console.WriteLine("MyArray의 크기 : " + MyArray.Length);
    }

    static int[ ]  CreateIntArray(int size)
    {
        int[ ]  intArray = new int[size];
        return intArray;
    }
}
```

Array의 활용

메소드의 인자로서의 배열

```
class ArrayParam
{
        static void Main(string[ ]  args)
        {
                int[] MyArray = {2,6,5,4,1};
                MyMethod(MyArray);
                // 3, 7, 6, 5, 2
                for(int i=0;i<MyArray.Length;i++)
                        Console.WriteLine (MyArray[i]);
        }
        static void MyMethod(int[ ]  parameter)
        {
                for(int j=0;j<parameter.Length;j++)
                        parameter[j]++;
        }
}
```

Array의 예제

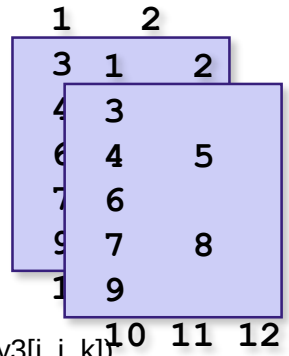
```
int[ ]  Array1 = new int[4] {2, 1, 4, 5};        // one-dimensional array
for (int i = 0; i < Array1.Length; i++)
    Console.WriteLine("Array1[{0}]={1}", i, Array1[i]);

int[,] Array2 = new int[2,3];                    // two-dimensional array
int a = 1;
for (int i = 0; i < Array2.GetLength(0); i++)
    for (int j = 0; j < Array2.GetLength(1); j++)
        Array2[i, j] = a++;
Array2[1, 2] = 7;

int[, ,] Array3 = new int[2, 4, 3] {             // three-dimensional array
                    { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } },
                    { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 }, { 10, 11, 12 } }
                    };
for (int i = 0; i < Array3.GetLength(0); i++)
        for (int j = 0; j < Array3.GetLength(1); j++)
            for (int k = 0; k < Array3.GetLength(2); k++)
                Console.WriteLine("Array3[{0},{1},{2}]={3}", i, j, k, Array3[i, j, k]);
```

2	1	4	5

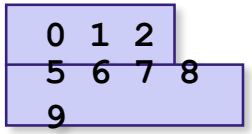
1	2	3
4	5	7



Array의 예제

```
int[ ] [ ]  Array4 = new int[2][ ] ;                    // jagged array
Array4[0] = new int[] { 0, 1, 2 };
Array4[1] = new int[] { 5, 6, 7, 8, 9 };

for (int i = 0; i < Array4.GetLength(0); i++)
        for (int j = 0; j < Array4[i].GetLength(0); j++)
            Console.WriteLine("Array4[{0}][{1}]={2}", i, j, Array4[i][j]);
```




```
int[ ] [ ]  Array5 = new int[ ] [ ]  {                  // jagged array
                new int[] {1, 2},
                new int[] {3, 4, 5},
                new int[] {6, 7, 8, 9},
                new int[] {10, 11, 12}
                };

for (int i = 0; i < Array5.Length; i++)
    for (int j = 0; j < Array5[i].Length; j++)
        Console.WriteLine("Array5[{0}][{1}]={2}", i, j, Array5[i][j]);
```