

C#프로그래밍의 이해

5주차

02. RETURN에 대하여 (1/2)

- ▶ return 문 (1/2)
 - ▶ 점프문의 한 종류
 - ▶ 프로그램의 흐름을 갑자기 호출자에게로 돌림
 - ▶ return 문은 언제든지 메소드 중간에 호출되어 메소드를 종결시키고 프로그램의 흐름을 호출자에게 돌려줄 수 있음

```
int Fibonacci( int n )  
{  
    if (n < 2)  
        return n;  
    else  
        return Fibonacci(n-1) + Fibonacci(n-2);  
}
```

재귀호출 : 메소드 내부에서 스스로를 다시 호출

02. RETURN에 대하여 (2/2)

▶ return 문 (2/2)

- ▶ 메소드가 반환할 것이 아무것도 없는 경우(즉, 반환 형식이 void인 경우)에도 return 문 사용 가능
- ▶ 이 경우 return문은 어떤 값도 반환하지 않고 메소드만 종료시킴

```
void PrintProfile(string name, string phone)
{
    if (name == "")
    {
        Console.WriteLine("이름을 입력해주세요.");
        return;
    }
    Console.WriteLine( "Name:{0}, Phone:{1}", name, phone );
}
```

03. 매개 변수에 대하여 (1/3)

- ▶ 메소드를 호출할 때 넘기는 매개 변수는 그대로 메소드 안으로 넘겨지는 것일까? : 답은 “아니다.”

메소드 선언부의 매개 변수

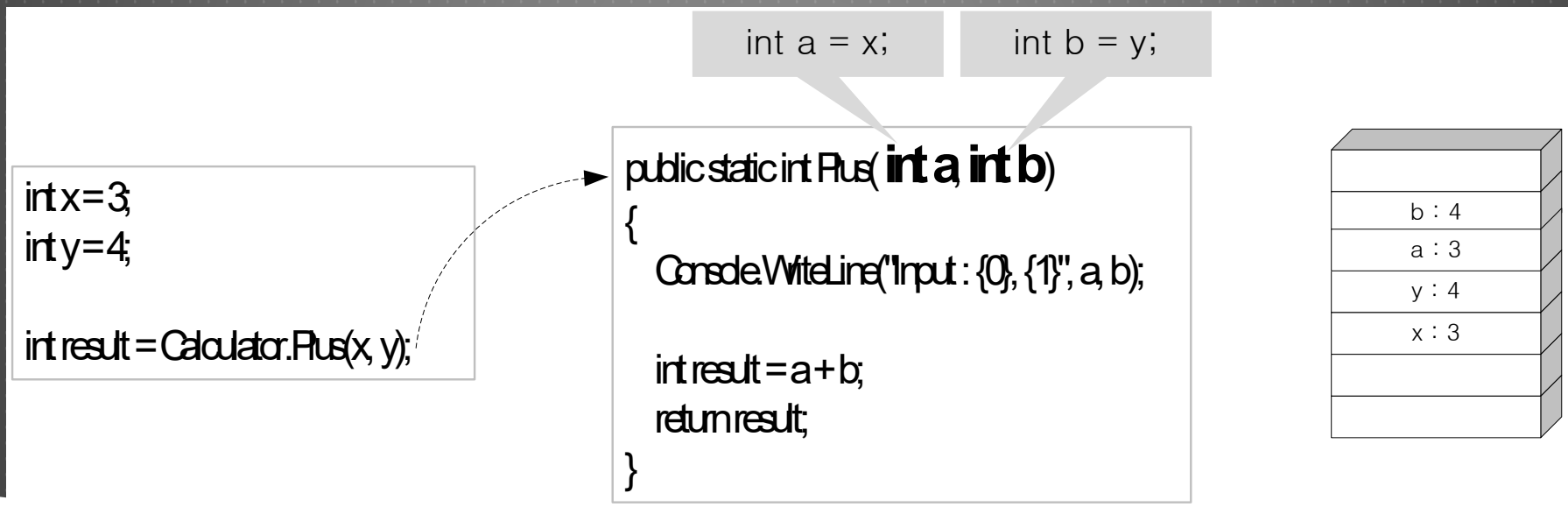
```
class Calculator
{
    public static int Plus(int a, int b)
    {
        Console.WriteLine("Input : {0}, {1}", a, b);
        int result = a + b;
        return result;
    }
    // ...
}
```

메소드 호출부의 매개 변수

```
class MainApp
{
    public static void Main()
    {
        int x = 3;
        int y = 4;
        int result = Calculator.Plus(x, y);
        // ...
    }
}
```

03. 매개 변수에 대하여 (2/3)

- ▶ 매개 변수도 메소드 외부에서 메소드 내부로 데이터를 전달하는 매개체 역할을 할 뿐, 근본적으로는 “변수”
 - ▶ 한 변수를 또 다른 변수에 할당하면 변수가 담고 있는 데이터만 복사됨.



03. 매개 변수에 대하여 (3/3)

- ▶ 예제 : 두 매개 변수의 값을 교환하는 Swap() 메소드

```
public static void Swap(int a, int b)
{
    int temp = b;
    b = a;
    a = temp;
}
```

```
static void Main(string[] args)
{
    int x = 3;
    int y = 4;

    Swap(x, y);
}
```

Swap() 함수를 호출하고난 뒤에도 x와 y의 값은 변하지 않는다.

04. 참조에 의한 매개 변수 전달 (1/2)

- ▶ 앞에서 작성했던 Swap() 메소드는 두 매개변수의 값을 교환하지 못함
 - 제대로 기능을 하는 Swap() 메소드를 만들 방법은?
- ▶ 참조에 의한 전달(Call by reference)
 - ▶ 값에 의한 전달이 매개 변수가 변수나 상수로부터 값을 복사하는 것과는 달리, 참조에 의한 전달은 매개 변수가 메소드에 넘겨진 원본 변수를 직접 참조
 - ▶ 메소드 선언과 호출 시 **ref** 키워드를 이용

```
static void Swap( ref int a, ref int b)
{
    int temp = b;
    b = a;
    a = temp;
}
//
```

```
int x = 3;
int y = 4;
```

```
Swap( ref x, ref y);
```

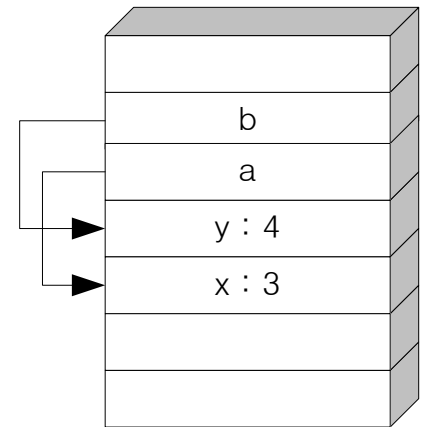
Swap() 함수를 호출하고난 뒤에 x는 4, y는 3으로 바뀐다.

04. 참조에 의한 매개 변수 전달 (2/2)

- ▶ Swap() 메소드가 참조로 매개 변수를 전달할 때의 과정

```
int x=3;  
int y=4;  
  
Swap(ref x, ref y);
```

```
static void Swap(ref int a, ref int b)  
{  
    int temp=b;  
    b=a;  
    a=temp;  
}
```



05. 출력 전용 매개 변수 (1/2)

- ▶ 대개의 경우 메소드의 결과는 하나면 충분하지만, 두 개 이상의 결과를 요구하는 코드를 작성해야 하는 경우가 생김.
 - ▶ 예) 나눗셈의 몫과 나머지를 모두 한번에 반환해야 하는 메소드
- ▶ `ref` 키워드를 이용하면 외부에서 넘긴 매개변수를 메소드 내부에서 변경할 수 있으므로 이러한 요구를 만족시킬 수 있음. 하지만...
 - ➔ `ref` 키워드는 메소드가 결과를 저장하지 않아도 아무런 에러를 일으키지 않음
 - ➔ 자신이나 다른 프로그래머가 코드를 다시 읽을 때 혼돈을 일으킬 소지가 있음.
- ▶ 그래서 C#은 **출력 전용 매개 변수**에 사용할 수 있도록 **out** 키워드 제공

05. 출력 전용 매개 변수 (2/2)

▶ 출력 전용 매개 변수의 선언 및 사용 예

```
void Divide( int a, int b, out int quotient, out int remainder )  
{  
    quotient = a / b;  
    remainder = a % b;  
}
```

```
int a = 20;  
int b = 3;  
int c;  
int d;
```

```
Divide( a, b, out c, out d );  
Console.WriteLine("Quotient : {0}, Remainder {1}", c, d );
```



06. 메소드 오버로딩

- ▶ Overloading : 과적하다
- ▶ 메소드 오버로딩 : 하나의 메소드 이름에 여러 개의 구현을 올림

```
int Plus(int a, int b)←
```

```
{  
  return a + b;  
}
```

```
double Plus(double a, double b) ←
```

```
{  
  return a + b;  
}
```

호출

```
int result1 = Plus( 1, 2 );
```

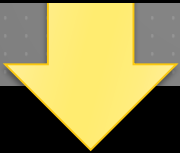
```
double result2 = Plus( 3.1, 2.4 );
```

호출

07. 가변길이 매개 변수 (1/2)

- ▶ 가변길이 매개변수 : 개수가 유연하게 변할 수 있는 매개 변수
- ▶ 다음 코드에서 호출하는 Sum() 메소드는 가변길이 매개 변수를 이용하여 단 하나만 구현(오버로딩을 이용하지 않음)

```
int total = 0;  
  
total = Sum( 1, 2 );  
total = Sum( 1, 2, 3 );  
total = Sum( 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 );
```



```
int Sum( params int[] args )  
{  
    int sum = 0;  
    for(int i=0; i<args.Length; i++)  
    {  
        sum += args[i];  
    }  
    return sum;  
}
```

가변길이 매개변수는 **params**
키워드와 **배열**을 이용하여 선언

07. 가변길이 매개 변수 (2/2)

- ▶ 메소드 오버로딩 vs 가변길이 매개 변수
 - ▶ 메소드 오버로딩
 - 매개 변수의 개수가 유한하게 정해져 있을 때 사용
 - 매개 변수의 각 형식이 다를 때 사용
 - ▶ 가변길이 매개 변수
 - 형식은 같으나 매개 변수의 개수만 유연하게 달라질 수 있는 경우에 사용

08. 명명된 매개 변수

- ▶ 메소드를 호출할 때 매개 변수 목록 중 어느 매개 변수에 데이터를 할당할 것인지를 지정하는 것은 "순서"
- ▶ 명명된 매개 변수(Named Parameter)는 메소드를 호출할 때 매개 변수의 이름을 명시함으로써 순서에 관계없이 매개 변수에 할당할 데이터를 바인드하는 기능

```
static void PrintProfile( string name, string phone)
{
    Console.WriteLine("Name:{0}, Phone:{1}", name, phone);
}
static void Main(string[] args)
{
    PrintProfile( name : "박찬호", phone : "010-123-1234");
}
```

09. 선택적 매개 변수

- ▶ 메소드 선언시 매개 변수에 기본 값을 할당함으로써, 해당 매개 변수에 명시적으로 값을 할당할지/않을지를 선택가능하게 하는 기능
- ▶ 선택적 매개 변수를 가지는 메소드 선언의 예

```
void MyMethod( int a = 0, int b = 0 )  
{  
    Console.WriteLine( "{0},{1}", a, b );  
}
```

b는 선택적 매개 변수

- ▶ 선택적 매개 변수의 호출 예

```
MyMethod(3);    // 매개변수 b 생략  
MyMethod(3, 4);
```

클래스

객체지향 프로그래밍과 클래스

01. 객체 지향 프로그래밍과 클래스 (1/2)

- ▶ 객체 지향 프로그래밍(Object Oriented Programming)
 - ▶ 코드 내의 모든 것을 객체(Object)로 표현하고자 하는 프로그래밍 패러다임
 - ▶ 객체 : 세상의 모든 것(사람, 연필, 자동차, 파일, 모니터, 상품 주문 등)을 지칭하는 단어
 - ▶ 실제 세상의 객체들을 어떻게 코드로 표현할까?
 - ➔ "추상화(抽象化:Abstraction)"를 통해 실제 객체의 주요한 특징들만 뽑아 C# 코드로 표현
 - ➔ 객체 지향 프로그래밍은 프로그래머의 추상화 능력을 필요로 함

01. 객체 지향 프로그래밍과 클래스 (2/2)

▶ 클래스

- ▶ 객체를 만들기 위한 청사진
- ▶ 붕어빵틀은 클래스, 붕어빵은 객체
- ▶ `int a = 3;` 에서 `int`는 클래스, `a`는 객체
- ▶ 객체(object)는 instance라고 부르기도 함. 인스턴스는 청사진의 실체라는 뜻
- ▶ 객체에게서 뽑아낸 속성과 기능은 클래스 안에 변수와 메소드로 표현
- ▶ 모든 클래스는 복합 데이터 형식임. 즉, 기본 데이터 형식을 조합해서 만드는 사용자 정의 데이터 형식.

02. 클래스의 선언과 객체의 생성 (1/2)

▶ 클래스 선언 형식

```
class 클래스이름  
{  
    // 데이터와 메소드  
}
```

▶ 클래스 선언 예

```
class Cat  
{  
    public string Name;  
    public string Color;  
  
    public void Meow()  
    {  
        Console.WriteLine("{0} : 야옹", Name);  
    }  
}
```

데이터

메소드

02. 클래스의 선언과 객체의 생성 (2/2)

▶ 클래스의 인스턴스, 즉 객체 생성 예

```
Cat kitty = new Cat();  
kitty.Color = "하얀색";  
kitty.Name = "키티";  
kitty.Meow();  
Console.WriteLine("{0} : {1}", kitty.Name, kitty.Color);
```

Cat()은 생성자

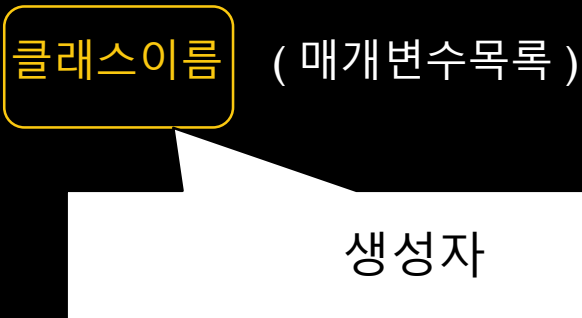
```
Cat nero = new Cat();  
nero.Color = "검은색";  
nero.Name = "네로";  
nero.Meow();  
Console.WriteLine("{0} : {1}", nero.Name, nero.Color);
```

03. 객체의 삶과 죽음에 대하여: 생성자와 소멸자 (1/4)

▶ 생성자 선언 형식

```
class 클래스이름
{
    한정자 클래스이름 (매개변수목록)
    {
        //
    }

    // 필드
    // 메소드
}
```



- ▶ 프로그래머가 명시적으로 생성자를 선언하지 않아도 컴파일러가 암시적으로 기본생성자를 제공해줌

03. 객체의 삶과 죽음에 대하여: 생성자와 소멸자 (2/4)

▶ 생성자 선언 예

```
class Cat
```

```
{
```

```
public Cat()
```

```
{
```

```
    Name = "";
```

```
    Color = "";
```

```
}
```

```
public Cat( string _Name, string _Color )
```

```
{
```

```
    Name = _Name;
```

```
    Color = _Color;
```

```
}
```

```
public string Name;
```

```
public string Color;
```

```
// ...
```

```
}
```

생성자는 객체의 초기화에 필요한 파라미터를 입력받을 수도 있음.

```
Cat kitty = new Cat();  
kitty.Name = "키티";  
kitty.Color = "하얀색";
```

```
Cat nabi = new Cat( "나비", "갈색" );
```

03. 객체의 삶과 죽음에 대하여: 생성자와 소멸자 (3/4)

▶ 소멸자 선언 형식

```
class 클래스이름  
{  
    ~클래스이름()  
    {  
        //  
    }  
  
    // 필드  
    // 메소드  
}
```

소멸자

- ▶ 프로그래머가 명시적으로 소멸자를 선언하지 않아도 컴파일러가 암시적으로 기본소멸자를 제공해줌

03. 객체의 삶과 죽음에 대하여: 생성자와 소멸자 (4/4)

- ▶ 소멸자를 직접 구현하지 않는 것이 좋은 이유
 - 1) 소멸자가 언제 호출될지 예측할 수 없다.
 - 2) 가비지 컬렉터가 상당히 똑똑하게 객체의 소멸을 처리한다.
- ▶ 소멸자를 직접 구현하는 것이 좋은 이유
 - 1) 해제하여야 할 중요 변수 및 메모리관련 내용이 있을 경우 객체를 소멸시킬 때 코드를 통해 해제할 수 있음
(가비지 컬렉터로 해제될 때 까지 기다리지 않아도 됨)

04. 객체 복사하기: 얇은 복사와 깊은 복사 (1/4)

- ▶ 아래 코드의 출력 결과는?

```
class MyClass
{
    public int MyField1;
    public int MyField2;
}

MyClass source = new MyClass();
source.MyField1 = 10;
source.MyField2 = 20;

MyClass target = source;
target.MyField2 = 30;

Console.WriteLine( "{0} {1}", source.MyField1, source.MyField2 );
Console.WriteLine( "{0} {1}", target.MyField1, target.MyField2 );
```

실행 결과 :

10 30
10 30

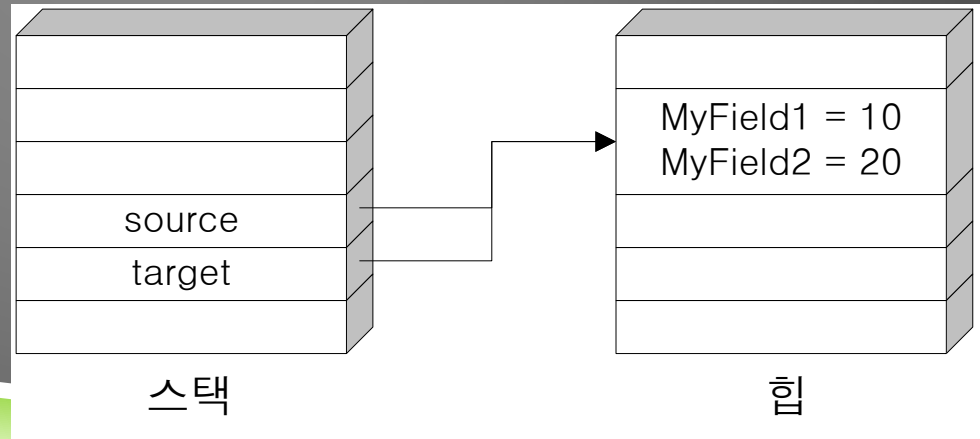
04. 객체 복사하기: 얇은 복사와 깊은 복사 (2/4)

- ▶ 왜 결과가 10 30 / 10 30이었을까?
 - ▶ 아래의 코드에서 target은 source의 전체 필드 값을 복사하는 것이 아닌, source가 참조하고 있는 힙의 주소만 복사해왔기 때문임

```
MyClass source = new MyClass();  
source.MyField1 = 10;  
source.MyField2 = 20;
```

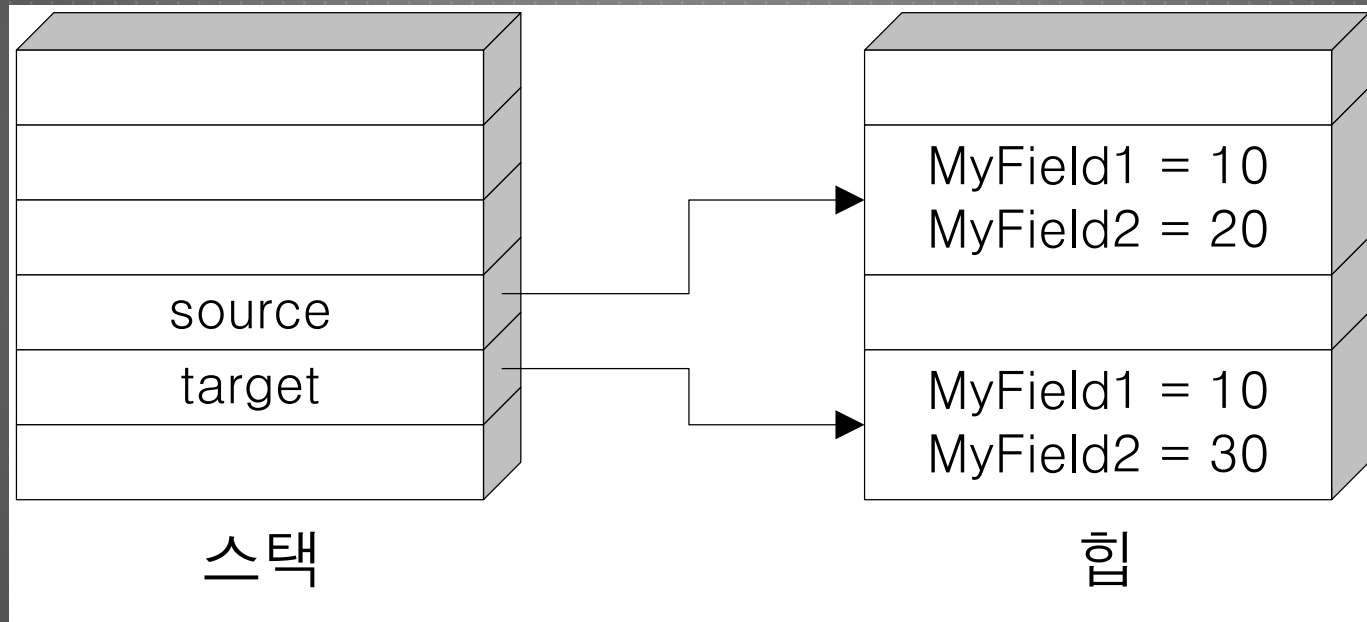
```
MyClass target = source;
```

- ▶ 이것을 “얇은 복사”라고 함.



04. 객체 복사하기: 얇은 복사와 깊은 복사 (3/4)

- ▶ 우리가 원하는 것은 “깊은 복사”
 - ▶ 원본 객체와 별도의 객체를 할당하여, 각 필드의 값을 복사해 넣는 것.



04. 객체 복사하기: 얇은 복사와 깊은 복사 (4/4)

- ▶ 깊은 복사를 위해서는 명시적으로 필드를 복사하는 코드가 필요

```
class MyClass
{
    public int MyField1;
    public int MyField2;

    public MyClass DeepCopy()
    {
        MyClass newCopy = new MyClass();
        newCopy.MyField1 = this.MyField1;
        newCopy.MyField2 = this.MyField2;
        return newCopy;
    }
}
// ...
```

```
MyClass source = new MyClass();
source.MyField1 = 10;
source.MyField2 = 20;
```

```
MyClass target = source.DeepCopy();
```

05. THIS 키워드

▶ this 키워드

- ▶ 객체가 스스로를 가리키는 키워드
- ▶ 객체 외부에서 객체의 필드나 메소드에 접근하기 위해 객체의 이름(변수 또는 식별자)를 사용하듯,
객체 내부에서는 자신의 필드나 메소드에 접근할 때 this 키워드 사용

```
class Employee
{
    private string Name;

    public void SetName( string Name )
    {
        this.Name = Name;
    }
}
```

