C#
-CAHPTER2-

SOUL SEEK

목차

연산자

1. 연산자

- 산술 연산자, 증감 연산자, 관계 연산자, 조건 연산자, 논리 연산자, 비트 연산자, 할당 연산자

분류	연산자	분류	연산자
산술 연산자	**+, -, *, /, %**	논리 연산자	**&&, ||, !**
증감 연산자	**++, --**	비트 연산자	**<<, >>, &, |, ^, ~**
관계 연산자	**<, >, ==, !=, <=, >=**	할당 연산자	**=, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=**
조건 연산자	**?:**		

- 연산자 우선 순위

순위	종류	연산자	순위	종류	연산자
1	증감 연산자	후위 **a++, a--**	**8**	비트 논리 연산자	**&**
2	증감 연산자	전위**++a, --a**	**9**	비트 논리 연산자	**^**
3	산술 연산자	***, /, %**	**10**	비트 논리 연산자	**|**
4	산술 연산자	**+, -**	**11**	논리 연산자	**&&**
5	시프트 연산자	**<<, >>**	**12**	논리 연산자	**||**
6	관계 연산자	**<, >, <=, >=, is, as**	**13**	조건 연산자	**?:**
7	관계 연산자	**==, !=**	**14**	할당 연산자	**=, *=, /=, %/, +=, -=, <<=, >>=, &=, ^=, |=**

1. 연산자

조건 연산자

- 조건식 **?** 참일 때의 값 **:** 거짓 일 때의 값**;**

int a = 30;
string result = a == 30 ? “참” : “거짓”;

비트 연산자(시프트 연산자, 비트 논리 연산자)

- **2**진수 비트 단위로 연산 하는 것을 비트 연산이라고 한다**.**
- 암호화**,** 복호화**,** 논리 연산을 이용한 다중 옵션 등등 으로 활용한다**.**

연산자	이름	설명	지원 형식
<<	왼쪽 시프트	첫 번째 피연산자의 비트를 두 번째 피연산자의 수만큼 왼쪽으로 이동시킨다**.**	첫 번째 피연산자는 **int, uint, long, ulong**이며 피연산자는 **int**형식만이 지원
>>	오른쪽 시프트	첫 번째 피연산자의 비트를 두 번째 피연산자의 수만큼 오른쪽으로 이동시킨다**.**	**<<**와 동일
&	논리곱 연산자	두 피연산자의 비트 논리곱을 수행한다**.**	정수 계열 형식과 **bool** 형식에 대해 사용랑 수 있다**.**
!	논리합 연산자	두 피연산자의 비트 논리합을 수행한다**.**	**&**와 동일

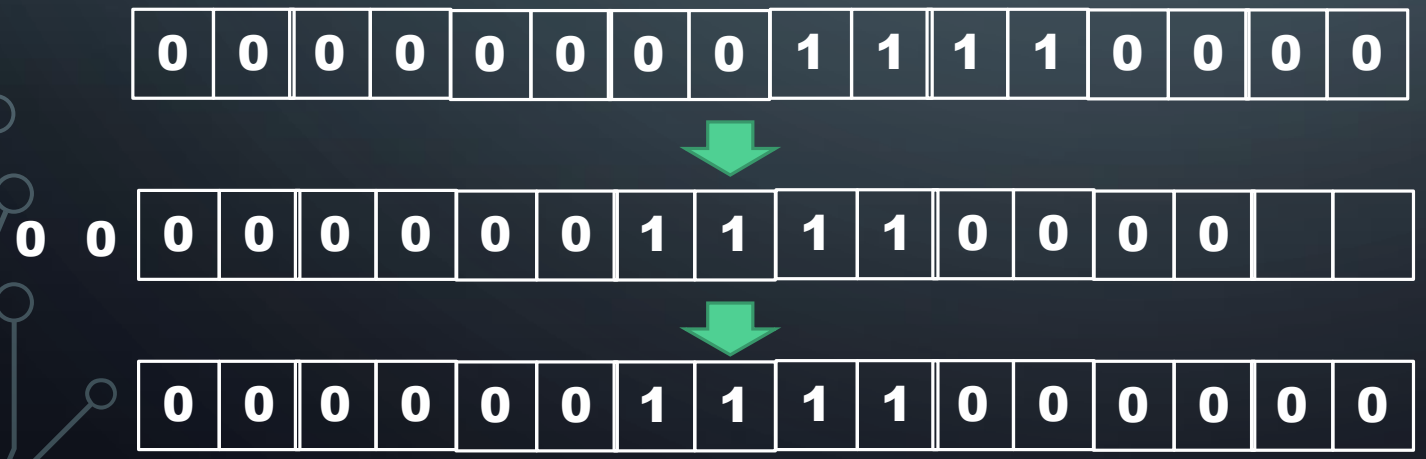
4. 연산자

연산자	이름	설명	지원형식
^	배타적 논리합 연산자	두 피연산자의 비트 배타적 논리합을 수행한다.	&와 동일
~	보수 연산자	피연산자의 비트를 0으로 1로, 1은 0으로 반전 시킵니다. 단항 연산자.	int, uint, long, ulong에 대해 사용이 가능하다.

시프트 연산자

- 비트를 왼쪽이나 오른쪽으로 이동 시킨다.
- 비트를 이동시킨 순서대로 다시 원래대로 돌려 놓는 것을 이용해 암/복호화로 응용한다.
- 고속의 곱연산을 실행할 때 활용한다.
- 원본 데이터를 **a** 옮긴 비트 수를 **b**라고 했을 때, 왼쪽 시프트는 **a * 2 ^ b,** 오른쪽 시프트는 **a / 2 ^ b**라는 결과가 나온다.

10진수 **240**을 비트 이동하여 보자



int형은 **32**비트 이지만 **16**비트로 표현해 보았다.

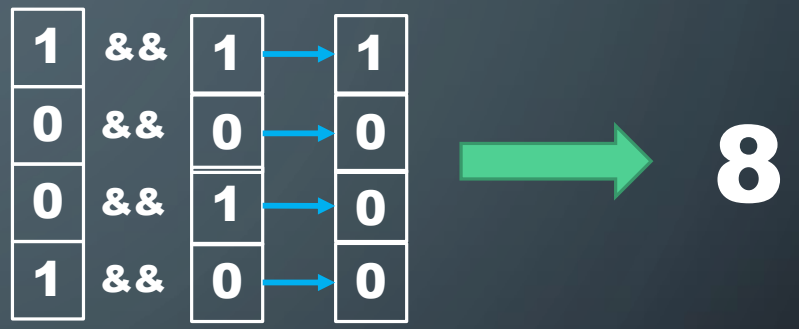
"비트를 이동(Shift)" 시킨다고 부르며 이동하면서 비트 각각의 숫자는 변화한 것이 없지만 이것을 원래 형태로 복원하게 되면 다르게 보일 것이다. 이런 특수성을 이용해 암/복호화에 사용한다.

4. 연산자

비트 논리 연산자

- 비트가 **0, 1**로 나타나기 때문에 이것을 논리 연산으로 **bool**로 반환하거나 뒤집는 것으로 논리 관계를 표현한다**.**
- 다중 옵션으로 사용했던 부분에서 비트 논리 연산자가 사용 되었다**.**
- **1**을 참 **0**을 거짓

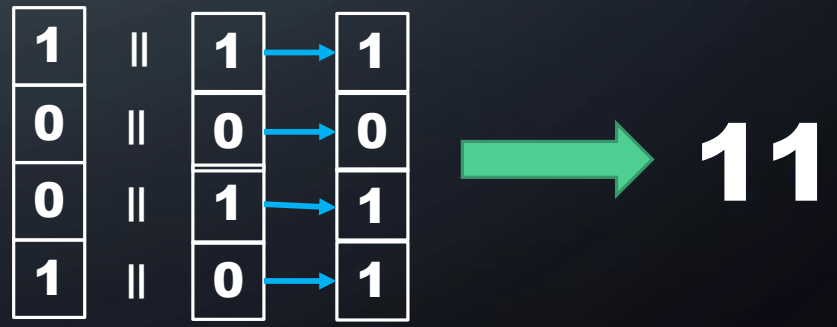
논리곱 연산자(&)
→두 비트 모두 참**(1)**이면 참**,** 이외의 경우는 거짓**(0)**



int result = 9 & 10; // result은 **8**

논리합 연산자(|)
→두 비트 중 하나라도 참**(1)**이면 참**,** 이외의 경우는 거짓

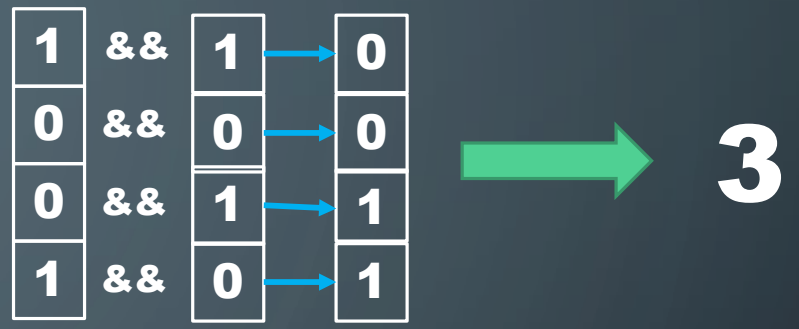
int result = 9 | 10; // result은 **11**



4. 연산자

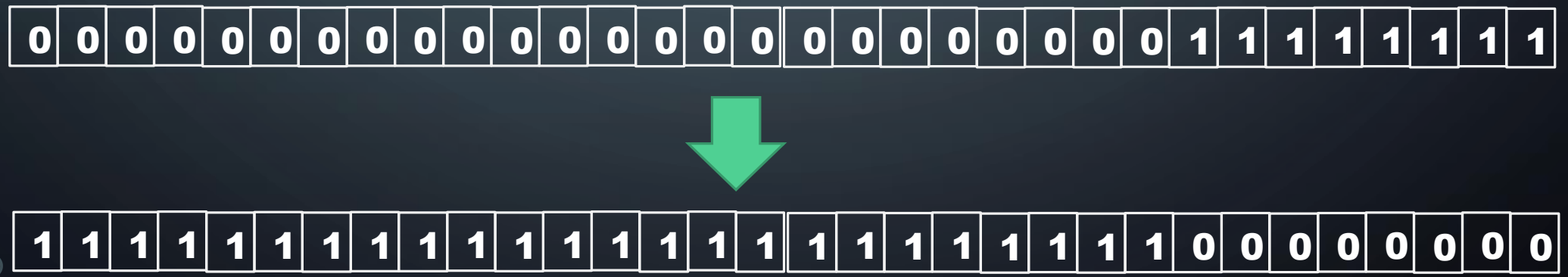
배타적 논리합 연산자(^)
→진리 값이 서로 달라야 참, 같으면 거짓

int result = 9 ^ 10; // result은 **3**



보수 연산자(~)
→단항 연산자
→반대 비트로 바꿔주는 것 **1→0, 0→1**로 변환 시키는 것

Int a = 255;
Int result = ~a; // result는 **-256;**



제어문/반복문

2. 제어문/반복문

제어문

- **if, else, else if, switch case: break;, continue;, goto, return, throw**

반복문

while, for, do while, foreach

foreach

- **while, for**에 비해 진행 속도가 느리다. – 치명적이지 않다.
- 메모리에 올라가는 힙 타입을 순차적으로 순회하기 가장 적합하다.
- 순서가 정해져 있지 않은 데이터들의 집합을 순회할 때 자주 사용된다.
- **var**을 많이 사용하게 된다.

foreach(데이터 형식 변수명 **in** 배열 또는 컬렉션**,** 제너릭**(**일반화**))**
{ 코드 또는 코드 블록**}**

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

foreach(int a in arr )              → foreach(var a in arr)
    Console.WriteLine(a);
```

METHOD

3. METHOD

Method

- **C/C++**에서는 **Function(**함수**)**라 부르며**,** 다양한 언어마다 부르는 명칭이 다르기 때문에 메소드라고 하면 함수로 생각하면 된다**.**
- **C/C++**에서 사용하는 **Function** 선언 규칙을 대부분 가지고있다**.** ➔ 오버로딩**,** 추상화 동일**.**
- 일련의 코드를 하나의 이름 아래 묶어서 같은 일은 하는 녀석이라고 지칭하게 한다**.**
- 각 언어의 규칙에 따라 명명하고 선언하는 순간 **Method**는 호출되고 이것을 **Method Call**이라고 부른다**.** ➔ 이 말에서 **Callback**의 의미를 상기시킬 수 있다**.**

```
class 클래스 이름
{
    한정자 반환형식 method의 이름( 매개 변수)
    {
        //실행 코드 1
        //실행 코드 2
            .
            .
        return method의 결과;
    }
}
```

Class 안에 **method**가 선언된다 **C#**은 객체지향 언어이고 모든 것을 객체로 표현한다**.**
각 객체는 각자의 데이터와 기능**(method)**를 갖고 있는데**, Class**가 이 객체들을 위한 청사진을 제공한다**.**

3. METHOD

매개변수

- 일반적인 **C/C++**에서 사용되는 매개변수 전달법과 다른 것은 없다.
- **참조에 의한 매개변수를 전달하는 방법(ref)**은 똑같이 사용된다.
- 포인터가 존재하지 않기 때문에 포인터 매개변수 대신 **출력 전용 매개 변수(out)을 활용**한다.
- 가변길이 매개변수라는 것이 존재한다.
- 선택적 매개 변수(디폴트 매개변수)는 동일하게 사용할 수 있다.

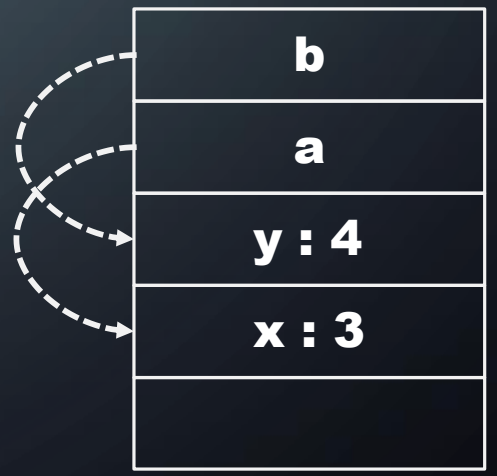
ref

- **C/C++**의 레퍼런스 매개변수와 같은 동작을 한다.
- 결과 값의 대입여부와 상관없이 동작한다.

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

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

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



3. METHOD

out

- 포인터 매개 변수와 같은 동작을 한다.
- 결과값을 대입해주지 않는다면 에러가 난다. 이는 연산하지 않고 초기화 되지 않은 상태의 쓰레기 값을 그대로 전달해 주지 않기 위한 안전장치이다.
- **ref**보다 **out** 사용을 권장한다.
  - ➔ **ref**를 이용해서 매개 변수를 넘기는 경우에는 메소드가 해당 매개 변수에 결과를 저장하지 않아도 컴파일러는 아무런 경고를 하지 않는다. 한편, **out**은 메소드를 호출하는 곳에서는 초기화를 하지 않은 지역 변수를 메소드의 **out** 매개 변수로 넘기는 것은 가능하지만 런타임 상황에서 컴파일 에러 메시지를 출력한다.
  - ➔ 컴파일러가 코드 작성시 바로 오류를 표시해주지 않고 런타임시 결정되는 상황이기 때문에 런타임 오류 체크가 훨씬 힘들다 그러므로 미리 안전장치가 있는 **out**을 사용을 권장한다.

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

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

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

3. METHOD

가변길이 매개 변수

- **params** 키워드와 배열을 이용해서 선언.
- 오버로딩의 경우 매개변수의 타입이 서로 다를 경우에는 유용하다 하지만 매개변수의 타입의 변화가 없는데 그 수가 늘어나서 적용하고 싶은 경우가 있다 이때 가변길이 매개변수를 사용 할 수 있다.

```
int sum(params int[] args) // 배열 형태로 담을 수 있다.
{
    int sum = 0;

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

    return sum;
}

Console.WriteLine(Sum(1, 2));
Console.WriteLine(Sum(1, 2, 3, 4));
```

3. METHOD

선택적 매개 변수(디폴트 매개변수)

- **C++**의 디폴트 매개변수 같은 기능을 하게 해줍니다.
- 이 선택적 매개변수는 매개변수를 다 선언하고 제일 뒤쪽에 자리해야 하며 선택적 매개변수의 수는 정해져 있지 않습니다.
- 메소드 오버로딩과 혼용해서 사용하지 않는 것이 좋다. 비슷해 보이지만 사용하는 응용에서 차이가 확실히 나기 때문이다.
- 값을 넘겨주지 않아도 기본 값 설정에 의해 특수한 상황에서만 값이 적용되는 경우를 만들고 싶다면 선택적 매개 변수(디폴트 매개 변수), 하나의 함수에서 다양한 경우의 수에 따라 다르게 동작하는 것을 만들어 주고 싶다면 오버로딩을 사용하면 된다.

```
int MyMethod(int a = 0, int b = 1)
{
    return a + b;
}

Console.WriteLine(MyMethod());
Console.WriteLine(MyMethod(1));
Console.WriteLine(MyMethod(1, 2));
```