C#
-CAHPTER1-

SOUL SEEK

목차

1. 기초이론
2. HelloWorld
3. 변수

기초 이론

1. 기초 이론

.NET Framework

- 여러 기능을 지원하는 클래스 라이브러리를 제공
- 프레임 워크를 설치하면 다양한 플렛폼을 지원하는 동작을 하게 된다.
- **C#**에 최적화 되어 있다.

CRL(Common Language Runtime)

- **Java JVM** 같은 가상 머신 기능을 한다.
- **.NET**프레임워크와 함께 **OS**위에 설치된다.
- **Native Code**로 작성된 프로그램들은 운영체제가 직접 실행할 수 있다.
- **C#**은 **OS**가 바로 알아볼 수 없는 **IL**이라는 중간 언어로 작성되어 있어 **JIT** 과정이 필요하다.

JIT(Just In Time) 컴파일

- **IL**이라는 중간 언어로 작성된 실행파일을 만들어낸다. 사용자가 이 파일을 실행시키면 **CLR**이 중간 코드를 읽어 들여서 다시 **OS**가 이해할 수 있는 **Native Code**로 컴파일한 후 실행하게 된다. 서로 다른 멀티 플렛폼을 지원하기 위한 과정이기 때문에 **C#**역시 이런 컨셉으로 만들어진 언어이기 때문이다. **C/C++**을 모두 포함하고 **Java** 특유의 문법 형식을 포함한 사실 **Window OS** 환경에서 사용할 수 있는 모든 라이브러리가 포함되기 때문에 빌드해서 로드하는 시간이 **C/C++**에 비해 오래 걸린다.
- 하지만 이런 부분은 단지 개발자가 조금의 시간을 감수하면 되는 부분이라 성능에 영향을 미치는 건 적다고 볼 수 있다.

HELLOWORLD

2. HELLOWORLD

```
using System;

namespace HelloWorld
{
    class HelloWorld
    {
        //CLR에 메모리 할당
        static void Main(string[] args)
        {
            //Hello World 출력
            Console.WriteLine("Hello World!!");

            //콘솔창 유지할려고 넣은 코드
            Console.ReadKey();
        }
    }
}
```

2. HELLOWORLD

- **C/C++**과 달리 **C#**은 **.h, .cpp**파일로 나뉘어져 생성되지 않고 **.cs** 단일 파일로 생성된다**.**
- 파일 단위가 클래스가 되며 파일명이 그대로 클래스명이 된다**.**
- 전역**,** 지역변수 개념은 없고 멤버 변수만 존재한다**.** → **static**을 이용한 공용화 가능**.**

using System;
➔ 키워드 선언 **# include <stdio.h>** 과 같은 역할을 한다**.**
➔ **C#**에서는 사용자가 제작한 클래스나 파일을 참조 할 경우 **.h**파일을 사용하지 않는다**.**

Console.WriteLine()
➔ 출력 함수**, printf()**와 같은 역할을 한다**.**

```
namespace HelloWorld
{
    구조체..
    클래스..
    인터페이스..
}
```

변수

3. 변수

- 초기화 필수 ➔ 초기화 없이 쓰레기 값을 가지게 되면 컴파일 에러 발생!
- 변수타입 : **Value Type, Reference Type**이 존재한다.
- **Reference Type : Heap**영역 ➔ **string, object**
  - 힙 메모리에 할당된 데이터들은 언제 까지나 살아 있다. 하지만 **C#**에서는 일정시간 사용하지 않으면 사용자가 실수로 해제하지 않은 것으로 간주하여 **GC**가 일어나면서 메모리에서 삭제한다. → **new**는 사용하지만 **delete**의 기능이 필요 없다.
- **Value Type : Stack**영역 ➔ 숫자, 정수, 문자(문자열 아님), 부동소수, 논리형식
  - 스택 메모리에 순차적으로 쌓아놓고 코드블록이 끝나는 지점에서 모두 사라진다.

- **정수 계열 형식**
  ➔ **C# 전용이 아닌 .Net Framework에 포함된 언어는 공용이다.**

데이터형식	설명	크기(바이트)	범위	데이터 형식	설명	크기	범위
byte	부호없는 정수	**1(8**비트**)**	**0~255**	**uint**	부호 없는 정수	**4(32**비트**)**	**42**억
sbyte	부호있는 정수	**1(8**비트**)**	**-128~127**	**long**	정수	**8(64**비트**)**	**-92**경**~92**경
short	정수	**2(16**비트**)**	**-32,768~32,767**	**ulong**	부호 없는 정수	**8(64**비트**)**	**184**경
ushort	부호 없는 정수	**2(16**비트**)**	**0~65,535**	**char**	유니코드 문자	**2(16**비트**)**	
int	정수	**4(32**비트**)**	**-21**억**~21**억				

3. 변수

- **부동 소수 형식**

데이터형식	설명	크기(바이트)	범위
float	단일 정밀도 부동 소수점 형식 **(7**개의 자릿수를 다룰 수 있음**)**	**4(32**비트**)**	**-3.402823e38 ~ 3.402823e38**
double	복수 정밀도 부동 소수점 형식 **(15~16**개의 자릿수를 다룰 수 있음**)**	**8(64**비트**)**	**-1.7976931348623e308 ~ 1.7976931348623e308**

string

- 어떤 물건이 가지런히 연속적으로 놓여 있는 줄이라는 뜻
  ➔ 문자가 연속적으로 가지런히 놓여 있는 줄**,** 문자 배열**,** 문자열**…** 이라는 의미가 된다**.**
- 문자열 간의 연산에 편리한 기능을 제공한다**.**
- **C++ string**과 비슷하지만 내부적으로 동작**,** 구성이 다르다**.** ➔ 기능적으로는 같다**.**
- 문자와 문자열의 대입법은 다르다**.**

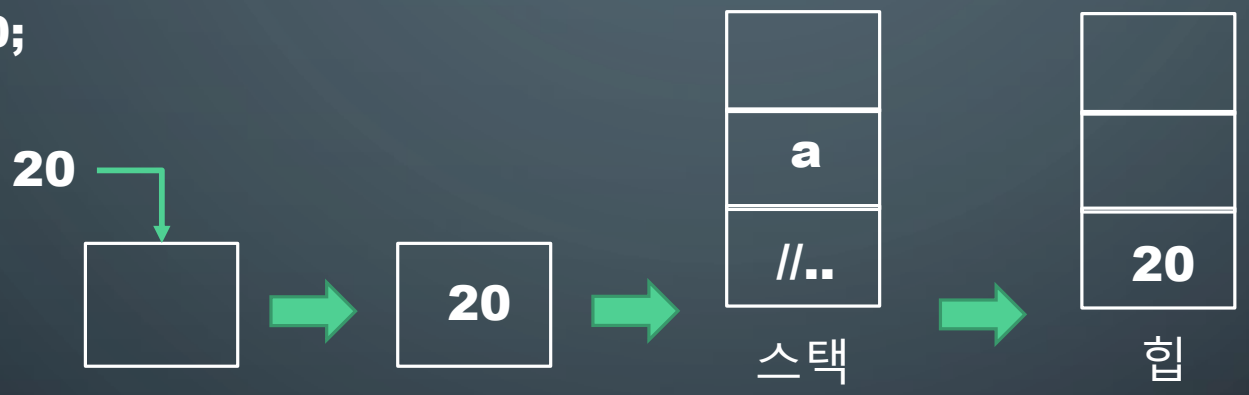
ex)
char c = ‘안**’;**
string s = “안녕하세요**”;**

3. 변수

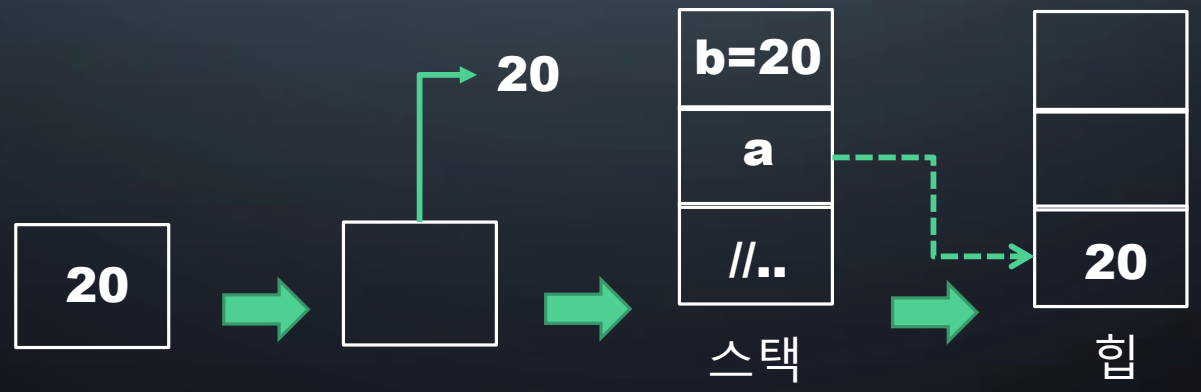
object

- 모든 데이터 형식을 대변할 수 있다.
- 모든 데이터 형식은 **object**를 상속받고 있기 때문에 어떠한 값이 대입되어도 맞는 형식으로 적용해준다.
- **Reference Type**이기 때문에 힙 메모리 영역에 저장된다.
- **object**로 실질적으로 선언된 실제 값들은 **heap**에 저장하고 **heap**에 저장된 값들의 주소 값들만 스택에 저장한다. 이런 과정에 의해서 박싱과 언박싱이 발생한다.

```
object a = 20;
```




```
object a = 20;
Int b = (int)a;
//int b = a;
```



3. 변수

박싱(Boxing)

- 값타입을 **Object** 형식 또는 이 값 형식에서 구현된 임의의 인터페이스 형식으로 변환하는 것을 말한다.
- 기존에 저장된 스택영역에서 힙영역에 값형식을 저장한다.

```
int i = 123;
object o = i; // 이 과정에서 박싱이 일어난다.
```

- **int**의 값형식을 **Object**라는 참조형식으로 형변환을 시도한다. 그 목적은 여러가지가 있을 수 있는데, 보통 파라미터로 전달되거나, **List<Object>** 형식으로 모든 값들을 입력 받는 목적으로도 사용될 수 있다.

- 메모리상에서 아래와 같이 메모리가 할당이 된다.

- **스택 영역에 있는 i 값이 o로 변환이 되면서 힙 영역에 object 형식으로 선언이 되고, 값이 복사된다.**
- **o는 스택 영역에 존재하며 boxed된 i의 주소 값을 가지고 있다.**
- **박싱은 보통 암시적으로 되며, 명시적으로도 가능하다.**
  → object o = a, object o = (int)a;

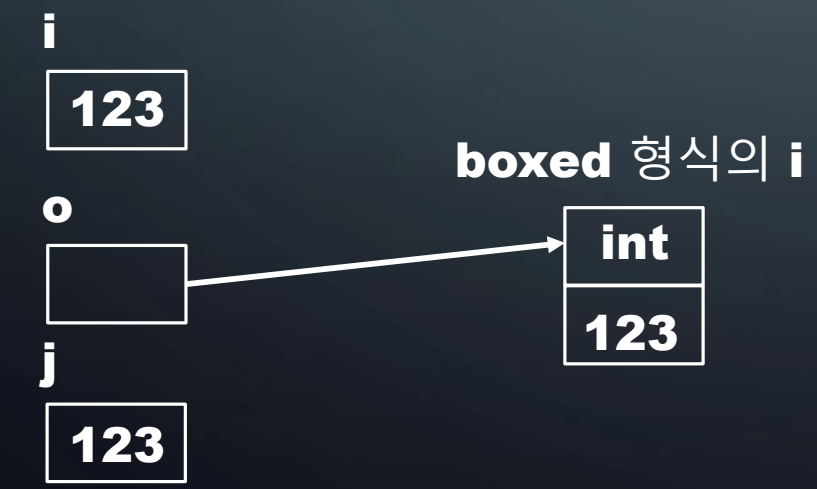
3. 변수

언박싱(UnBoxing)

- **Object** 형식에서 값형식으로, 또는 인터페이스 형식에서 해당 인터페이스를 구현하는 값 형식으로 변환하는 것을 말한다.

```
int i = 123;
object o = i;
int j = (int)o; //언박싱 과정이 일어난다.
```

언 박싱 과정
1.개체 인스턴스가 지정한 값 형식을 boxing한 값인지 확인
2. 인스턴스의 값을 값 형식 변수에 복사



- **int**를 박싱한 **o**의 객체를 다시 **int** 타입의 **j** 값에 넣고 있다.
- 언박싱을 할 때 다른 타입으로 하거나, 해당 타입보다 작은 범위로 변환을 하려면 오류**(InvalidCastException)**가 발생한다.
- 이때는 미리 같은 타입인지를 먼저 확인하는 절차를 거쳐야한다.
- **is** 연산자를 이용해서 미리 같은 타입인지 확인 후에 캐스팅을 해야 안전하다.

3. 변수

박싱, 언박싱 주의 사항

- **박싱을 하면 단순히 참조에 할당하는 것보다 20배까지 시간이 소모되며, 언박싱은 할당에 4배정도 소모된다.**
- 박싱과 언박싱에는 많은 시간이 소모된다. 되도록이면 제네릭을 사용해서 박싱과 언박싱이 일어나지 않도록 구성을 해야 하며, 어쩔 수 없이 사용하려면 그 타입에 맞는 캐스팅을 해서 오류가 없이 처리해야한다.

형 변환 - 문자열을 숫자로, 숫자를 문자열로

- 숫자를 문자열로..

```
int c = 1234;
string d = (string)c;
string e = c.ToString();
```

- 문자를 숫자로..

```
int a = int.Parse(“1234”);
string d = “1234”;
int b = int.Parse(d);
```

3. 변수

var 형식

- 지역변수로만 사용 가능하다.
- 컴파일러가 값이 담긴 형식을 찾아서 자동으로 맞는 형식으로 지정해준다.
- 컴파일 하면서 적용되기 때문에 컴파일을 진행하기 이전까지는 오류체크가 힘들다.
- 컴파일시 결정되는 상황이기때문에 약간 속도 이슈가 있지만 빌드 상황이 끝난 후에는 문제가 되지 않는다.

var a = 3; // a는 int형으로..
var b = “Hello” // b는 strin형으로..