HCI 프로그래밍

6. Interface

HCI
Human Computer Interaction

Multiple Inheritance

- 다중 상속 **(Multiple Inheritance)**란 하나의 클래스가 여러 개의 클래스로부터 상속을 받는 것
  - **C#에서는 하나의 클래스가 동시에 2개 이상의 클래스에서 상속 받는 것(Multiple Inheritance)을 지원하지 않음**
  - C#에서는 인터페이스(interface)를 이용하여 다중상속 지원

Interfaces

Interface

- 행동적인 특성(메소드, 속성, 인덱서, 이벤트) 만을 정의
- 인터페이스는 하위 클래스에 구현되어야 하는 기능을 선언할 시 일반적으로 이질적인 클래스들이 공통으로 제공해야 할 메소드들을 선언할 때 사용
- 다중상속을 구현하기 위한 방법으로 사용
- `class` 대신 `interface` 키워드를 사용하여 선언

관례적으로 "I" 접두어 사용

public →

```
interface IToken : IBase
{
    int LineNumber( );
    string Name( );
}
```

← *Interface 상속*

인터페이스의 메소드는 구현부분이 없이 ';'으로 처리

Class & Interfaces

클래스/인터페이스 관계

- **클래스 & 클래스 : 상속관계**
- **클래스 & 인터페이스 : 구현관계**
- **인터페이스 & 인터페이스 : 상속관계**

클래스 & 인터페이스 선언 예

```
interface IMyInterface: IBase1, IBase2 {
    void MethodA();
    void MethodB();
}

class ClassA: IFace1, IFace2
{ // class members , implementing interface }

class ClassB: BaseClass, IFace1, IFace2
{ // class members, implementing interface }
```

여러 개의 인터페이스를 상속 받은 인터페이스는 상위 인터페이스의 메소드를 모두 구현해 주어야 함

클래스와 인터페이스를 함께 상속 받는 경우 기본 클래스가 처음에 위치

Class & Interfaces

```
//interface
using System;
interface IScalable {
    void ScaleX(float factor);
    void ScaleY(float factor);
}
public abstract class DrawObject {
    public DrawObject() {}
    public abstract void Print();
}
public class TextObject: DrawObject,
IScalable {
    private string text;
    public TextObject(string text) {
        this.text = text;
    }
    // implemention of IScalable.ScaleX()
    public void ScaleX(float factor) {
        Console.WriteLine("ScaleX: {0} {1}", text,
                          factor);
    }
    // implemention of IScalable.ScaleY()
    public void ScaleY(float factor) {
        Console.WriteLine("ScaleY: {0} {1}", text,
                          factor);
    }
    // implemention of Print()
    public override void Print() {
        Console.WriteLine("TextObject: {0}",
text);
    }
}
```

Class & Interfaces

```
// TextObject 클래스 객체를 array로 처리
class  InterfaceTest {
  public static void Main()  {
    DrawObject[] dArray =
            new  DrawObject[10];

    dArray[0] = new TextObject("Text1");
    dArray[1] = new TextObject("Text2");

 /* array gets initialized here, with classes that
derive from DiagramObject. Some of them
implement IScalable. */
```

```
    foreach (DrawObject d in dArray)
    {
      if (d is IScalable)  {
        IScalable scalable = (IScalable) d;
        scalable.ScaleX(0.1F);
        scalable.ScaleY(10.0F);
        d.Print();
      }
    }
  }
}
```

```
ScaleX: Text1 0.1
ScaleY: Text1 10
TextObject: Text1

ScaleX: Text2 0.1
ScaleY: Text2 10
TextObject: Text2
```

IComparable 인터페이스

- 현 객체를 동일한 형식의 다른 객체와 비교하는 (인터페이스 정렬 순서를 확인하는) ComparableTo 메소드를 제공하는 인터페이스
- int CompareTo (Object obj) obj 인수는 인터페이스를 구현하는 클래스와 같은 형식 리턴 값은 현 객체가 obj보다 작으면 0보다 작은 수를, 같으면 0을, 크면 0보다 큰 수를 리턴하도록 구현

```
public class Age : IComparable {
        public int CompareTo(object obj) {
                if (obj is Age) {
                        Age temp = (Age) obj;
                        return m_value.CompareTo(temp.m_value);
                } throw new ArgumentException("Object is not Age");
        }
        protected int m_value;
}
```

IEquatable

IEquatable 인터페이스

- 현 객체가 동일한 형태의 다른 객체와 같은지 여부를 확인하는 Equals 메소드를 제공하는 인터페이스
- bool Equals (Object obj) obj 인수는 인터페이스를 구현하는 클래스와 같은 형식 리턴 값은 현 객체가 obj과 같으면 true, 다르면 false를 리턴하도록 구현

```
public class Person : IEquatable<Person> {
        public bool Equals(Person p) {
            if (p is Person) {
                return name.Equals(p.name) && age.Equals(p.age);
            } throw new ArgumentException("Object is not Person");
        }
        public string name;
        public int age;
}
```

IEnumerable

IEnumerable 인터페이스

- Collections을 반복하는 열거자 (IEnumerator 객체)를 반환하는 GetEnumerator 메소드를 제공하는 인터페이스
- IEnumerator GetEnumerator ()

IEnumerator 인터페이스

- 컬랙션의 요소들을 참조할 수 있는 아래 3 메소드 및 속성을 갖는 인터페이스
- object Current – 컬랙션의 현재 요소를 가져오는 속성
- bool MoveNext() – 컬랙션의 다음 요소로 이동하는 메소드
- void Reset() – 컬랙션의 첫 번째 요소 앞의 초기 위치로 설정하는 메소드

IEnumerable

```
//IEnumerable interface 구현
using System;
Using System.Collections;

public class Tokens : IEnumerable {
   private string[] elements;
   Tokens(string source, char[] delimiters) {
      elements = source.Split(delimiters);
   }
   // implemention of GetEnumerator()
   public IEnumerator GetEnumerator() {
      return new TokenEnumerator(this);
   }
   // implemention of TokenEnumerator
   private class TokenEnumerator :
IEnumerator {
      private int position = -1;
      private Tokens t;
      public TokenEnumerator(Tokens t) {
         this.t = t;
      }
      public void Reset() {
         position = -1;
      }
      public bool MoveNext() {
         if (position < t.elements.Length – 1) {
            position++;
            return true;
         }
         else {
            return false;
         }
      }
      public object Current {
         get { return t.elements[position]; }
      }
   }
}
```

IEnumerable

```
// Token 테스트
class  EnumeratorTest {
  public static void Main()  {
    Tokens f = new Tokens("This is a sample
                  test.", new char [] {' ', '-'});
    foreach (string item in f) {
      Console.WriteLine(item);
    }
  }
}
```

Interface vs. Abst Class

인터페이스와 추상클래스의 공통점과 차이점

- 모두 추상의 의미
- new 키워드를 사용하여 객체를 생성하는 것이 불가능
- 파생클래스에서 모든 메서드를 구현하였을 경우 기능을 발휘
- 클래스와 메서드가 abstract로 선언되어 있다면 인터페이스로 변환가능

Interface vs. Abst Class

변환 예(추상클래스 → 인터페이스)

```
// 추상클래스
abstract public class StarPlayer {
    public abstract void GoodPlay();
    public abstract void Handsome();
}
```

```
// 인터페이스
interface IStarPlayer {
    void GoodPlay();
    void Handsome();
}
```

- 모든 추상클래스가 인터페이스로 될 수 없음
- 추상 메서드는 override 키워드 사용
- 추상 클래스는 다중상속을 지원하지 않음

Interface의 암시적 구현

```
public interface IGunner {
        void Shoot();
}
public interface ISoccerPlayer {
        void Shoot();
}
public class Gunner : IGunner {
        public void Shoot() { Console.WriteLine("총쏘기"); }
}
public class SoccerPlayer : ISoccerPlayer {
        public void Shoot() { Console.WriteLine("공차기"); }
}
// Main () 함수 중간생략
        Gunner g = new Gunner();
        g.Shoot();              // 총쏘기
        SoccerPlayer s = new SoccerPlayer();
        s.Shoot();              // 공차기
```

Interface의 명시적 구현

```
public class GunPlayer : IGunner, ISoccerPlayer {
        void IGunner.Shoot() { Console.WriteLine("총쏘기"); }
        void ISoccerPlayer.Shoot() { Console.WriteLine("공차기"); }
}
// Main () 함수 중간생략
        GunPlayer p = new GunPlayer();
        ((IGunner)p).Shoot();                  // 총쏘기 - IGunner로 형변환
        ((ISoccerPlayer)p).Shoot();            // 공차기 – ISoccerPlayer로 형변환

        // 또는 아래와 같이 해당 객체를 만들고 난 후 참조하도록 하는 방법을 사용
        IGunner g = new GunPlayer();           // IGunner로 형변환
        g.Shoot();                             // 총쏘기
        ISoccerPlayer s = new GunPlayer();     // ISoccerPlayer로 형변환
        s.Shoot();                             // 공차기
```

Default Interface Method

C# 8.0에서 추가된 Default interface method

- 코드 실행 블록을 가지고 있는 메소드로, 자바의 인터페이스 default method와 유사함
- Default interface 메소드를 사용하여 해당 인터페이스의 기존 구현과 소스 또는 호환성을 손상 시키지 않고 이후 버전의 인터페이스에 메서드를 추가할 수 있음
- 클래스는 인터페이스의 멤버를 상속하지 않음

```
interface IA {
   void M() {  Console.WriteLine("IA.M");  } // 기본 인터페이스 메소드
}
class B : IA { } // M() 메소드를 구현 안해도 됨
class C : IA { // default method를 새로 정의
   public void M() {  Console.WriteLine("C.M");  }
}
IA i = new B(); i.M(); // IA.M 기본 인터페이스 메소드 구현 사용
IA j = new C(); j.M(); // C.M
//new B().M(); // error: 클래스 B는 멤버 M()이 없음
new C().M(); // C.M
```

Default Interface Method

Default interface method는 구체적인 재정의가 필요

```
interface IA {
   void M() {  Console.WriteLine("IA.M");   }
}
interface IB : IA {
   void IA.M() {  Console.WriteLine("IB.M");   } // default method override
}
interface IC : IA {
   void IA.M() {  Console.WriteLine("IC.M");   } // default method override
}
interface ID : IB, IC {
}
class D : ID {
   void IA.M() {  Console.WriteLine("D IA.M");   } // default method override
}
D d = new D();
((IA)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((IB)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((IC)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((ID)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
//d.M(); // error: 클래스 D는 멤버 M()이 없음
```

Default Interface Method

```
interface IA {
    void M() {  Console.WriteLine("IA.M");   }
}
interface IE : IA {
    abstract void IA.M(); // 추상으로 재정의 가능
}
/*class E : IE { // error: IA.M()을 구현해야 함
}*/
class E : IE {
    void IA.M() {  Console.WriteLine("E IA.M");   } // default method override
}
IE ie = new E();
ie.M(); // 최종 override된 기본 인터페이스 메소드 사용 E IA.M
IA ia = new E();
ia.M(); // 최종 override된 기본 인터페이스 메소드 사용 E IA.M
```

Default Interface Method

```
interface IA {
    void M() {  Console.WriteLine("IA.M");   }
}
interface IE : IA {
    abstract void IA.M(); // 추상으로 재정의 가능
}
/*abstract class F : IE { // error: IA.M()을 구현해야 함
}*/
abstract class F : IE {
    void IA.M() {  Console.WriteLine("F IA.M");   } // default method override
}
class G : F {
}
G g = new G();
//g.M(); // error: 클래스 G는 멤버 M()이 없음
//((F)g).M(); // error: 클래스 F는 멤버 M()이 없음
((IE)g).M(); // 최종 override된 기본 인터페이스 메소드 사용 F IA.M
((IA)g).M(); // 최종 override된 기본 인터페이스 메소드 사용 F IA.M
//((IB)g).M(); // run-time error: InvaidCastException g는 IB로 변환될수 없음
```

Base/Derived Conversion :Casting

Upcasting

- 하위 클래스가 상위클래스로의 암시적인 형 변환

Downcasting

- 상위클래스가 하위클래스로의 형 변환 (downcasting) 시 명시적 형 변환 (explicit type conversion) 연산자 필요
- 실행 시 참조변수가 가리키는 실제 객체의 데이터 형 검사
- 실패하는 경우 "InvalidCastException" 발생

```
class UpcastDowncastTest {
   static void Main(string[] args) {
      MyBaseClass c = new MyBaseClass();
      MyDerivedClass d = new MyDerivedClass();
      c = d;                          // upcasting
      d = (MyDerivedClass) c;         // downcasting
   }
}
```

is & as Operator

is 연산자

- 데이터의 형 변환이 가능하면 true 반환

as 연산자

- 객체 사이의 형 변환 연산자
- 오류 발생시 exception 발생 없이 null 반환

```
Bird b;
if (a is Bird)
    b = (Bird) a;   // 안전한 형 변환
else
    Console.WriteLine("Not a Bird");

Bird b = a as Bird; // 형 변환
if (b == null)
    Console.WriteLine("Not a bird");
```

Object Type Conversion

모든 참조형(예, class)은 System.Object으로부터 파생

- 즉, 모든 참조형은 System.Object으로 형 변환이 가능

```
class Bird { … }
class Parrot : Bird { … }
class ObjectTypeConversionTest {
    static void Main(string[] args) {
        Bird b = new Bird();
        Parrot p = new Parrot();
        object o = p;              // object 형 변환
        Bird b2 = o as Bird;       // object형을 Bird 형으로 변환
        if (b2 == null)
            Console.WriteLine("Object is not a bird");
        else
            Console.WriteLine("Object is a bird");
    }
}
```

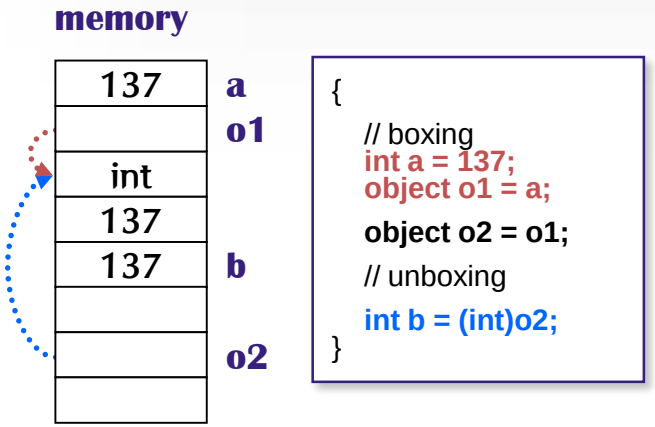
Boxing and Unboxing

Boxing

- 값형 -> 참조형으로 바꾸는 것 (암시적 변환)

Unboxing

- 참조형 -> 값형으로 바꾸는 것 (명시적 변환)



User-Defined Type Conversion

- 클래스나 구조체의 형을 다른 클래스, 구조체 혹은 기본 자료형으로 변환 가능
- 형변환 연산자**(conversion operator)**를 정의

```
class Sample {
    int number = 42;
    public static implicit operator string (Sample op) {
        return op.ToString();
    } // 문자열로 형변환하는 연산자를 정의했다면 ToString 메소드도 정의해야함
    public override string ToString () {
        return "Object: value=" + number;
    } // System.Object.ToString 메소드 재정의
}
// 형변환
Sample obj;
string s = obj; // implicit대신 explicit으로 선언되면 string s = (Sample)obj;
```

값 형과 참조 형의 비교

- 값형 경우 ==와 != 연산자를 사용하여 값을 비교
- 참조형 경우 ==와 != 연산자를 사용하여 비교할 수 없음

```
class Point { public int  x; public int y; }
class PointClassComparisonTest {
    static void Main() {
        int i = 1; int j = 1;       // 값형의 비교는 i==j는 true
        Point p1 = new Point();
        Point p2 = new Point();
        Point p3 = p1;                  // 같은 객체를 참조하므로 p1==p3는 true
        p1.x = 1; p1.y = 1;
        p2.x = 1; p2.y = 1;
        if (p1 == p2)              // 참조형의 비교는 p1==p2는 false
                Console.WriteLine("p1과 p2가 같다");
        else
                Console.WriteLine("p1과 p2가 다르다");
    }
}
```

Equals & Equality Ope

- 참조형 경우 비교 연산자를 사용할 수 있도록 **Equals(..) overriding**과 **== operator**와 **!= operator**를 **overloading**함

```
class Point: IEquatable<Point> {
    public int  x; public int y;
    public Point(): this(0, 0) {}
    public Point(int x, int y) { this.x = x; int.y = y; }
    // operator== overload
    public static bool operator==(Point p1, Point p2) { return p1.Equals(p2); }
    // operator!= overload
    public static bool operator!=(Point p1, Point p2) { return !p1.Equals(p2); }
    public override int GetHashCode() { return x ^ y; }
    public override bool Equals(object obj)
{
        if (!(obj is Point))
            return false;
        return Equals((Point)obj);
    }
```

Equals & Equality Operator Overloading

```
// IEquatable
    public bool Equals(Point other)
    {
        if (this.x == other.x && this.y == other.y)
            return true;
        return false;
    }
} // end of Point class
class PointClassComparisonTest {
    static void Main(string[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(3, 4);
        Point p3 = p1;
        if (p1 == p2)     // Equals와 ==operator로 p1==p2는 true
            Console.WriteLine("p1과 p2가 같다");
        if (p1 == p3)     // 같은 객체를 참조하므로 p1==p3는 true
            Console.WriteLine("p1과 p3가 같다");
    }
}
```

Method Overloading

메소드 오버로딩(Overloading)

- 한 클래스 내에서 두 개 이상의 이름이 같은 메소드 작성
  - ✓ 메소드 이름이 동일하여야 함
  - ✓ 매개 변수의 개수가 서로 다르거나, 타입이 서로 달라야 함
  - ✓ 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례
class MethodOverloading {
        public int Sum(int i, int j) {
                return i + j;
        }
        public int Sum(int i, int j, int k) {
                return i + j + k;
        }
        public double Sum(double i, double j) {
                return i + j;
        }
}
```

```
// 메소드 오버로딩이 실패한 사례
class MethodOverloadingFail {
        public int Sum(int i, int j) {
                return i + j;
        }
        public double Sum(int i, int j) {
                return (double)(i + j);
        }
}
```

Method Overriding

메소드 오버라이딩(Method Overriding)

- **슈퍼 클래스의 메소드를 서브 클래스에서 재정의**
  - ✓ **슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 타입 등 모든 것 동일하게 작성**
    - 이 중 하나라도 다르면 메소드 오버라이딩 실패
- **동적 바인딩 발생**
  - ✓ **서브 클래스에 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 바인딩 됨**