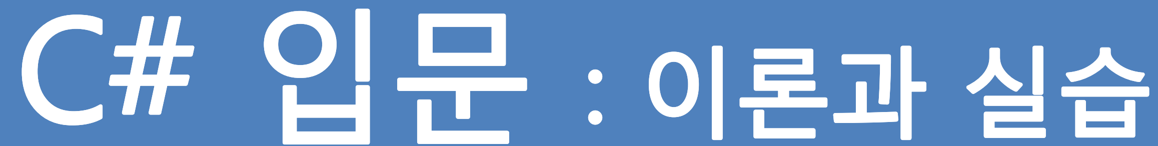
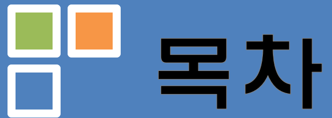


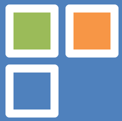
제 5장 클래스의 고급 사용

Programming Language Lab
Dongguk University



목차

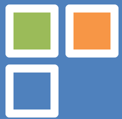
- 프로퍼티
- 인덱서
- 연산자 중복
- 인터페이스



프로퍼티 [1/4]

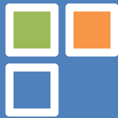
- 프로퍼티(property)
  - 클래스의 private 필드를 형식적으로 다루는 일종의 메소드.
  - 셋-접근자 - 값을 지정
  - 겟-접근자로 - 값을 참조
  - 겟-접근자 혹은 셋-접근자만 정의할 수 있음.
- 프로퍼티의 정의 형태

```
[property-modifiers] returnType PropertyName {
        get {
                // get-accessor body
        }
        set {
                // set-accessor body
        }
}
```



프로퍼티 [2/4]

- 프로퍼티 수정자
  - 수정자의 종류와 의미는 메소드와 모두 동일
  - 접근 수정자(4개), new, static, virtual, sealed, override, abstract, extern – 총11개
- 프로퍼티의 동작
  - 필드처럼 사용되지만, 메소드처럼 동작.
  - 배정문의 왼쪽에서 사용되면 셋-접근자 호출.
  - 배정문의 오른쪽에서 사용되면 겟-접근자 호출.

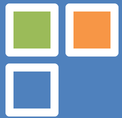


프로퍼티 [3/4]

[예제 5.1 - PropertyApp.cs]

```
using System;
class Fraction {
    private int numerator;
    private int denominator;
    public int Numerator {
        get { return numerator; }
        set { numerator = value; }
    }
    public int Denominator {
        get { return denominator; }
        set { denominator = value; }
    }
    override public string ToString() {
        return (numerator + "/" + denominator);
    }
}
class PropertyApp {
    public static void Main() {
        Fraction f = new Fraction(); int i;
        f.Numerator = 1;            // invoke set-accessor in Numerator
        i = f.Numerator+1;          // invoke get-accessor in Numerator
        f.Denominator = i;          // invoke set-accessor in Denominator
        Console.WriteLine(f.ToString());
    }
}
```

실행 결과 :
1/2

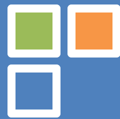


프로퍼티 [4/4]

[예제 5.3 - PropertyWithoutFieldApp.cs]

```
using System;
class PropertyWithoutFieldApp {
    public string Message {
        get { return Console.ReadLine(); }
        set { Console.WriteLine(value); }
    }
    public static void Main() {
        PropertyWithoutFieldApp obj = new PropertyWithoutFieldApp();
        obj.Message = obj.Message;
    }
}
```

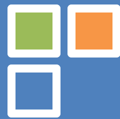
입력 데이터 :
    Hello
실행 결과 :
    Hello



인덱서 [1/2]

- 인덱서(indexer)
  - 배열 연산산자인 '[]'를 통해서 객체를 다룰 수 있도록 함
  - 지정어 this를 사용하고, '[]'안에 인덱스로 사용되는 매개 변수 선언.
  - 겟-접근자 혹은 셋-접근자만 정의할 수 있음.
- 인덱서의 수정자
  - static만 사용할 수 없으며, 의미는 메소드와 모두 같음.
  - 접근 수정자(4개), new, virtual, override, abstract, sealed, extern – 총10개
- 인덱서의 정의 형태

```
[indexer-modifiers] returnType this[parameterList] {
      set {
            // indexer body
      }
      get {
            // indexer body
      }
}
```



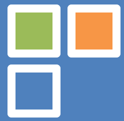
인덱서 [2/2]

[예제 5.5 - IndexerApp.cs]

```
using System;
class Color {
    private string[] color = new string[5];
    public string this[int index] {
        get { return color[index]; }
        set { color[index] = value; }
    }
}
class IndexerApp {
    public static void Main() {
        Color c = new Color();
        c[0] = "WHITE";  c[1] = "RED";
        c[2] = "YELLOW";  c[3] = "BLUE";
        c[4] = "BLACK";
        for(int i = 0 ; i < 5 ; i++)
            Console.WriteLine("Color is " + c[i]);
    }
}
```

실행 결과 :

```
Color is WHITE
Color is RED
Color is YELLOW
Color is BLUE
Color is BLACK
```

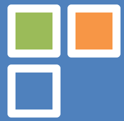


연산자 중복 [1/6]

- 연산자 중복의 의미
  - 시스템에서 제공한 연산자를 재정의 하는 것
  - 클래스만을 위한 연산자로서 자료 추상화가 가능
  - 문법적인 규칙은 변경 불가(연산 순위나 결합 법칙 등)

- 연산자 중복이 가능한 연산자

종 류	연 산 자
단 항	+, -, !, ~, ++, --, true, false
이 항	+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=
형 변환	변환하려는 자료형 이름

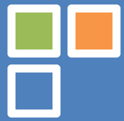


연산자 중복 [2/6]

- 연산자 중복 방법
  - 수정자는 반드시 public static.
  - 반환형은 연산자가 계산된 결과의 자료형.
  - 지정어 operator 사용, 연산기호로는 특수 문자 사용.

- 연산자 중복 정의 형태

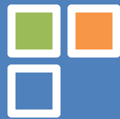
```
public static [extern] returnType operator op (parameter1 [, parameter2]) {
        // ... operator overloading body ...
}
```



연산자 중복 [3/6]

- 연산자 중복 정의 규칙

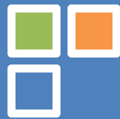
연 산 자	매개변수 형과 반환형 규칙
단항 +, -, !, ~	매개변수의 형은 자신의 클래스, 복귀형은 모든 자료형이 가능함.
++ / --	매개변수의 형은 자신의 클래스, 복귀형은 자신의 클래스이거나 파생 클래스이어야 함.
true / false	매개변수의 형은 자신의 클래스, 복귀형은 bool 형 이어야 함.
shift	첫 번째 매개변수의 형은 클래스, 두 번째 매개변수의 형은 int 형, 복귀형은 모든 자료형이 가능함.
이 항	shift 연산자를 제외한 이항 연산자인 경우, 두 개의 매개변수 중 하나는 자신의 클래스이며, 복귀형은 모든 자료형이 가능함.



연산자 중복 [4/6]

- 대칭적 방식으로 정의
  - true와 false, ==과 !=, <과 >, <=과 >=
- 형 변환 연산자(type-conversion operator)
  - 클래스 객체나 구조체를 다른 클래스나 구조체 또는 C# 기본 자료형으로 변환
  - 사용자 정의 형 변환(user-defined type conversion)
- 형 변환 연산자 문법 구조

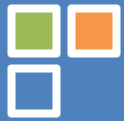
```
public static [extern] explicit operator type-name(parameter1)
        또는
public static [extern] implicit operator type-name(parameter1)
```



연산자 중복 [5/6]

[예제 5.7 - OperatorOverloadingApp.cs]

```
using System;
class Complex {
    private double realPart;              // 실수부
    private double imagePart;             // 허수부
    public Complex(double rVal, double iVal) {
        realPart = rVal;
        imagePart = iVal;
    }
    public static Complex operator+(Complex x1, Complex x2) {
        Complex x = new Complex(0, 0);
        x.realPart = x1.realPart + x2.realPart;
        x.imagePart = x1.imagePart + x2.imagePart;
        return x;
    }
    override public string ToString() {
        return "("+ realPart + "," + imagePart + "i)";
    }
}
```



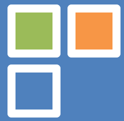
연산자 중복 [6/6]

[예제 5.7 - OperatorOverloadingApp.cs] – [계속]

```
class OperatorOverloadingApp {
    public static void Main() {
        Complex c, c1, c2;
        c1 = new Complex(1, 2);
        c2 = new Complex(3, 4);
        c = c1 + c2;
        Console.WriteLine(c1 + " + " + c2 + " = " + c);
    }
}
```

실행 결과 :

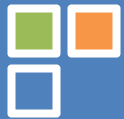
```
(1,2i) + (3,4i) = (4,6i)
```



인터페이스 [1/3]

- 인터페이스의 의미
  - 사용자 접속을 기술할 수 있는 프로그래밍 단위.
  - 구현되지 않은 멤버들로 구성된 수수한 설계의 표현.

- 인테페이스의 특징
  - 지정어 interface 사용.
  - 멤버로는 메소드, 프로퍼티, 인덱스, 이벤트가 올 수 있으며 모두 구현부분이 없음.
  - 다중 상속 가능.
  - 접근수정자 : public, protected, internal, private, new



인터페이스 [2/3]

- 인터페이스 선언 형태

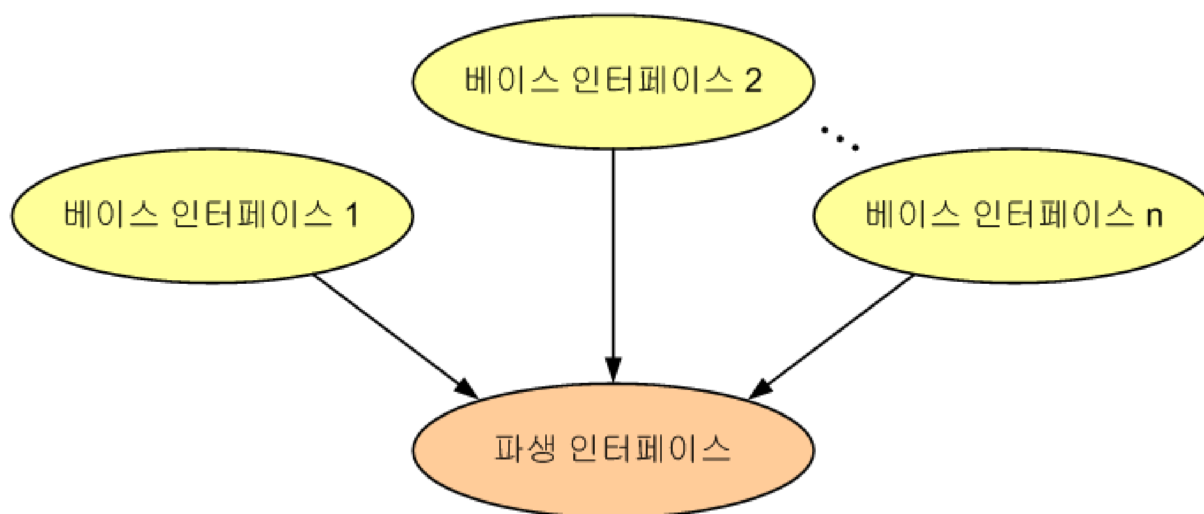
```
[interface-modifiers] [partial] interface InterfaceName {
        // interface body
}
```

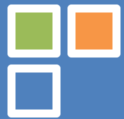
- 인터페이스 확장 형태

```
[modifiers] interface InterfaceName : ListOfBaseInterfaces {
        // method declarations
        // property declarations
        // indexer declarations
        // event declarations
}
```

인터페이스 [3/3]

- 인터페이스의 다중 상속



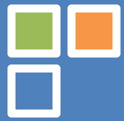


인터페이스 구현 [1/5]

- 인터페이스 구현 규칙
  - 인테페이스에 있는 모든 멤버는 묵시적으로 public이므로 접근수정자를 public으로 명시.
  - 멤버 중 하나라도 구현하지 않으면 derived 클래스는 추상클래스가 됨.

- 인테페이스 구현 형태

```
[class-modifiers] class ClassName : ListOfInterfaces {
        // member declarations
}
```



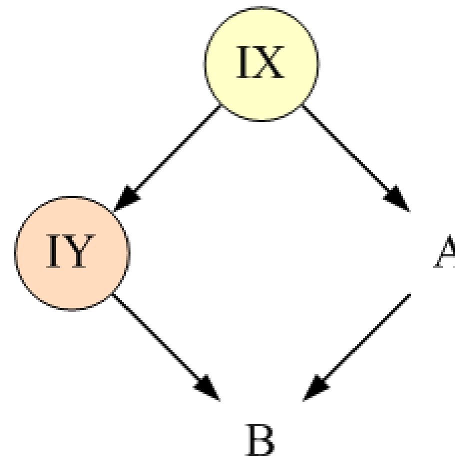
인터페이스 구현 [2/5]

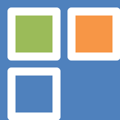
- 클래스 확장과 동시에 인터페이스 구현

```
[class-modifiers] class ClassName : BaseClass, ListOfInterfaces {
        // member declarations
}
```

- 다이아몬드 상속

```
interface IX { }
interface IY : IX { }
class A : IX { }
class B : A, IY { }
```

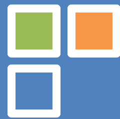




인터페이스 구현 [3/5]

[예제 5.10 - ImplementingInterfaceApp.cs]

```
using System;
interface IRectangle {
    void Area(int width, int height);
}
interface ITriangle {
    void Area(int width, int height);
}
class Shape : IRectangle, ITriangle {
    void IRectangle.Area(int width, int height) {
        Console.WriteLine("Rectangle Area : "+width*height);
    }
    void ITriangle.Area(int width, int height) {
        Console.WriteLine("Triangle Area : "+width*height/2);
    }
}
```



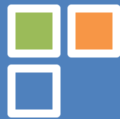
인터페이스 구현 [4/5]

[예제 5.10 - ImplementingInterfaceApp.cs] – [계속]

```
class ImplementingInterfaceApp {
    public static void Main() {
        Shape s = new Shape();
        // s.Area(10, 10);                    // error - ambiguous !!!
        // s.IRectangle.Area(10, 10);         // error
        // s.ITriangle.Area(10, 10);          // error
        ((IRectangle)s).Area(20, 20);         // 캐스팅-업
        ((ITriangle)s).Area(20, 20);          // 캐스팅-업
        IRectangle r = s;                     // 인터페이스로 캐스팅-업
        ITriangle t = s;                      // 인터페이스로 캐스팅-업
        r.Area(30, 30);
        t.Area(30, 30);
    }
}
```

실행 결과 :

```
Rectangle Area = 400
Triangle Area = 200
Rectangle Area = 900
Triangle Area = 450
```



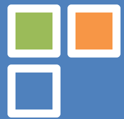
인터페이스 구현 [5/5]

[예제 5.11 - DiamondApp.cs]

```
using System;
interface IX { void XMethod(int i); }
interface IY : IX { void YMethod(int i); }
class A : IX {
    private int a;
    public int PropertyA {
        get { return a; }    set { a = value; }
    }
    public void XMethod(int i) { a = i; }
}
class B : A, IY {
    private int b;
    public int PropertyB {
        get { return b; }    set { b = value; }
    }
    public void YMethod(int i) { b = i; }
}
class DiamondApp {
    public static void Main() {
        B obj = new B();
        obj.XMethod(5);    obj.YMethod(10);
        Console.WriteLine("a = {0}, b = {1}",
                          obj.PropertyA, obj.PropertyB);
    }
}
```

실행 결과 :

```
a = 5, b = 10
```



인터페이스와 추상 클래스

- 공통점
  - 객체를 가질 수 없음
- 차이점
  - 인터페이스
    - 다중 상속 지원
    - 오직 메소드 선언만 가능
    - 메소드 구현 시, override 지정어를 사용할 수 없음
  - 추상 클래스
    - 단일 상속 지원
    - 메소드의 부분적인 구현 가능
    - 메소드 구현 시, override 지정어를 사용할 수 있음