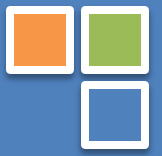


C# 입문 : 이론과 실습

제 4장 클래스

목차

클래스 [1/6]

- 클래스(Class)
  - C# 프로그램의 기본 단위
    - 재사용성(reusability), 이식성(portability), 유연성(flexibility) 증가
  - 객체를 정의하는 템플릿
    - 객체의 구조와 행위를 정의하는 방법
  - 자료 추상화(data abstraction)의 방법

- 객체(Object)
  - 클래스의 인스턴스로 변수와 같은 역할
  - 객체를 정의하기 위해서는 해당하는 클래스를 정의

클래스 [2/6]

- 클래스의 선언 형태

public, internal,
abstract, static, sealed

```
[class-modifier]  class  ClassName {
        // member declarations
}
```

필드, 메소드, 프로퍼티,
인덱서, 연산자 중복, 이벤트

클래스 [3/6]

- 수정자(modifier)
  - 부가적인 속성을 명시하는 방법
- 클래스 수정자(class modifier) – 8개
  - public
    - 다른 프로그램에서 사용 가능
  - internal
    - 같은 프로그램에서만 사용 가능
    - 수정자가 생략된 경우
  - static
    - 클래스의 모든 멤버가 정적 멤버
    - 객체 단위로 존재하는 것이 아니라 클래스 단위로 존재
  - abstract, sealed - 파생 클래스(4.2절)에서 설명
  - protected, private – 4.1.2절 참조
  - new – 중첩 클래스에서 사용되며 베이스 클래스의 멤버를 숨김

클래스 [4/6]

- 클래스 선언의 예
  - Fraction – 클래스형
    - 필드 2개, 메소드 계통의 멤버 3개

```
class Fraction {                    // 분수 클래스
    int numerator;                  // 분자 필드
    int denominator;                // 분모 필드

    public Fraction Add(Fraction f) { /*  ...  */ }                    // 덧셈 메소드
    public static Fraction operator+(Fraction f1, Fraction f2)        // 덧셈 연산자
    { /*  ...  */ }
    public void PrintFraction() { /*  ...  */ }                       // 출력 메소드
}
```

클래스 [5/6]

- 객체 선언
  - 클래스형의 변수 선언
  - 예) Fraction f1, f2;
    - f1, f2 – 객체를 참조(reference)하는 변수 선언
- 객체 생성
  - f1 = new Fraction();
  - Fraction f1 = new Fraction();
- 생성자
  - 객체를 생성할 때 객체의 초기화를 위해 자동으로 호출되는 루틴
  - 클래스와 동일한 이름을 갖는 메소드
  - 4.1.5절 참고

클래스 [6/6]

- 객체의 멤버 참조
  - 객체 이름과 멤버 사에에 멤버 접근 연산자인 점 연산자(dot operator) 사용
  - 예)

    필드 참조: f1.numerator

    메소드 참조: f1.Add(f2)

    연산자 중복: 직접 수식 사용 - 5.3절 참고
  - 멤버의 참조 형태

```
objectName.MemberName
```

필드

- 필드(field)
  - 객체의 구조를 기술하는 자료 부분
  - 변수의 선언으로 구성
- 필드 선언 형태

```
[field-modifier] DataType
fieldNames;
```

- 필드 선언 예

```
int anInterger, anotherInteger;
public string usage;
static long idNum = 0;
public static readonly double earthWeight = 5.97e24;
```

접근 수정자 [1/4]

- 접근 수정자(access modifier)
  - 다른 클래스에서 필드의 접근 허용 정도를 나타내는 속성

접근 수정자	동일 클래스	파생 클래스	네임스페이스	모든 클래스
private	O	X	X	X
protected	O	O	X	X
internal	O	X	O	X
protected internal	O	O	O	X
public	O	O	O	O

  - 접근 수정자의 선언 예

```
private int privateField;          // private
int noAccessModifier;              // private
protected int protectedField;      // protected
internal int internalField;        // internal
protected internal int piField;    // protected internal
public int publicField;            // public
```

접근 수정자 [2/4]

- private
  - 정의 된 클래스 내에서만 필드 접근 허용
  - 접근 수정자가 생략된 경우

```
class PrivateAccess {
    private int iamPrivate;
    int iamAlsoPrivate;
    // ...
}

class AnotherClass {
    void AccessMethod() {
        PrivateAccess pa = new PrivateAccess();
        pa.iamPrivate = 10;          // 에러
        pa.iamAlsoPrivate = 10;   // 에러
    }
}
```

접근 수정자 [3/4]

- public
  - 모든 클래스 및 네임스페이스에서 자유롭게 접근

```
class PublicAccess {
    public int iamPublic;
        // ...
}

class AnotherClass {
    void AccessMethod() {
        PublicAccess pa = new PublicAccess();
        pa.iamPublic = 10;            // OK
    }
}
```

접근 수정자 [4/4]

- internal
  - 같은 네임스페이스 내에서 자유롭게 접근
  - 네임스페이스 – 4.4절 참고
- protected
  - 파생 클래스에서만 참조 가능 - 4.2절 참고
- protected internal 또는 internal protected
  - 파생 클래스와 동일 네임스페이스 내에서도 자유롭게 접근

new / static 수정자 [1/2]

- new
  - 상속 계층에서 상위 클래스에서 선언된 멤버를 하위 클래스에서 새롭게 재정의하기 위해 사용

- static
  - 정적 필드(static field)
  - 클래스 단위로 존재
  - 생성 객체가 없는 경우에도 존재하는 변수
  - 정적 필드의 참조 형태

```
ClassName.staticField
```

new / static 수정자 [2/2]

[예제 4.2 - StaticVsInstanceApp.cs]

```
using System;
public class StaticVsInstanceApp {
    int instanceVariable;
    static int staticVariable;
    public static void Main() {
        StaticVsInstanceApp obj = new StaticVsInstanceApp();
        obj.instanceVariable = 10;                                  // ok
        //StaticVsInstanceApp.instanceVariable = 10;                // error
        StaticVsInstanceApp.staticVariable = 20;                    // ok
        //obj.staticVariable = 20;                                  // error
        Console.WriteLine("instance variable={0}, static variable={1}",
                obj.instanceVariable, StaticVsInstanceApp.staticVariable);
    }
}
```

실행 결과 :

```
instance variable=10, static variable=20
```

readonly / const 수정자

- readonly
  - 읽기전용 필드
  - 값이 변할수 없는 속성
  - 실행 중에 값에 값이 결정

- const
  - 값이 변할수 없는 속성
  - 컴파일 시간에 값이 결정
  - 상수 멤버의 선언 형태

```
[const-modifiers] const DataType constNames;
```

메소드

- 객체의 행위를 기술하는 방법
  - 객체의 상태를 검색하고 변경하는 작업
  - 특정한 행동을 처리하는 프로그램 코드를 포함하고 있는 함수의 형태

```
[method-modifiers] returnType MethodName(parameterList) {
        // method body
}
```

- 메소드 선언 예

```
class MethodExample {
    int SimpleMethod() {
        //...
    }
    public void EmptyMethod() { }
}
```

메소드 수정자

- 메소드 수정자: 총 11개
- 접근 수정자: public, protected, internal, private
- static
  - 정적 메소드
  - 전역 함수와 같은 역할
  - 정적 메소드는 해당 클래스의 정적 필드 또는 정적 메소드만 참조 가능
  - 정적 메소드 호출 형태

```
ClassName.MethodName();
```

- abstract / extern
  - 메소드 몸체 대신에 세미콜론(;)이 나옴
  - abstract – 메소드가 하위 클래스에 정의
  - extern – 메소드가 외부에 정의
- new, virtual, override, sealed – 4.2절 참조

매개변수 [1/2]

- 매개변수
  - 메소드 내에서만 참조될 수 있는 지역 변수
- 매개변수의 종류
  - 형식 매개변수(formal parameter)
    - 메소드를 정의할 때 사용하는 매개변수
  - 실 매개변수(actual parameter)
    - 메소드를 호출할 때 사용하는 매개변수
- 매개변수의 자료형
  - 기본형, 참조형

```
void parameterPass(int i, Fraction f) {
        // ...
}
```

매개변수 [2/2]

- 클래스 필드와 매개변수를 구별하기 위해 this 지정어 사용
  - this 지정어 - 자기 자신의 객체를 가리킴

```
class Fraction {
    int numerator, denominator;
    public Fraction(int numerator, int denominator) {
        this.numerator = numerator;
        this.denominator = denominator;
    }
}
```

매개변수 전달 [1/4]

- 값 호출(call by value)
  - 실 매개변수의 값이 형식 매개변수로 전달 – 예제 4.4
- 참조 호출(call by reference)
  - 주소 호출(call by address)
  - 실 매개변수의 주소가 형식 매개변수로 전달
  - C#에서 제공하는 방법
    - 매개변수 수정자 이용 – 예제 4.5
    - 객체 참조를 매개변수로 사용 – 예제 4.6
- 매개변수 수정자
  - ref – 매개변수가 전달될 때 반드시 초기화
  - out – 매개변수가 전달될 때 초기화하지 않아도 됨

매개변수 전달 [2/4]

[예제 4.4 - CallByValueApp.cs]

```
using System;
class CallByValueApp {
    static void Swap(int x, int y) {
        int temp;
        temp = x; x = y; y = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
    }
    public static void Main() {
        int x = 1, y = 2;
        Console.WriteLine("Before: x = {0}, y = {1}", x, y);
        Swap(x, y);
        Console.WriteLine(" After: x = {0}, y = {1}", x, y);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
 After: x = 1, y = 2
```

매개변수 전달 [3/4]

[예제 4.5 - CallByReferenceApp.cs]

```
using System;
class CallByReferenceApp {
    static void Swap(ref int x, ref int y) {
        int temp;
        temp = x; x = y; y = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
    }
    public static void Main() {
        int x = 1, y = 2;
        Console.WriteLine("Before: x = {0}, y = {1}", x, y);
        Swap(ref x, ref y);
        Console.WriteLine(" After: x = {0}, y = {1}", x, y);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
 After: x = 2, y = 1
```

매개변수 전달 [4/4]

[예제 4.6 - CallByObjectReferenceApp.cs]

```
using System;
class Integer {
    public int i;
    public Integer(int i) {
        this.i = i;
    }
}
class CallByObjectReferenceApp {
    static void Swap(Integer x, Integer y) {
        int temp = x.i; x.i = y.i; y.i = temp;
        Console.WriteLine("  Swap: x = {0}, y = {1}",x.i,y.i);
    }
    public static void Main() {
        Integer x = new Integer(1);  Integer y = new Integer(2);
        Console.WriteLine("Before: x = {0}, y = {1}",x.i,y.i);
        Swap(x, y);
        Console.WriteLine(" After: x = {0}, y = {1}",x.i,y.i);
    }
}
```

실행 결과 :

```
Before: x = 1, y = 2
 Swap: x = 2, y = 1
  After: x = 2, y = 1
```

매개변수 배열 [1/2]

- 매개변수 배열(parameter array)
  - 실 매개변수의 개수가 상황에 따라 가변적인 경우
    - 메소드를 정의할 때 형식 매개변수를 결정할 수 없음

- 매개변수 배열 정의 예

```
void ParameterArray1(params int[] args)   { /* ... */ }
void ParameterArray2(params object[] obj) { /* ... */ }
```

- 호출 예

```
ParameterArray1();
ParameterArray1(1);
ParameterArray1(1, 2, 3);
ParameterArray1(new int[] {1, 2, 3, 4});
```

매개변수 배열 [2/2]

[예제 4.7 - ParameterArrayApp.cs]

```
using System;
class ParameterArrayApp {
    static void ParameterArray(params object[] obj) {
        for (int i = 0; i < obj.Length; i++)
            Console.WriteLine(obj[i]);
    }
    public static void Main() {
        ParameterArray(123, "Hello", true, 'A');
    }
}
```

실행 결과 :

```
123
Hello
True
A
```

범용 메소드 [1/2]

- 범용 메소드(generic method)
  - 형 매개변수(type parameter)를 갖는 메소드
- 범용 메소드 정의 예

```
void Swap<DataType>(DataType x, DataType y) {
        DataType temp = x;
        x = y;
        y = temp;
}
```

- 범용 메소드 호출 예

```
Swap<int>(a, b);          // a, b: 정수형
Swap<double>(c, d);       // c, d: 실수형
```

범용 메소드 [2/2]

[예제 4.8 - GenericMethodApp.cs]

```
using System;
class GenericMethodApp {
    static void Swap<DataType>(ref DataType x, ref DataType y) {
        DataType temp;
        temp = x; x = y; y = temp;
    }
    public static void Main() {
        int a = 1, b = 2; double c = 1.5, d = 2.5;
        Console.WriteLine("Before: a = {0}, b = {1}", a, b);
        Swap<int>(ref a, ref b);            // 정수형 변수로 호출
        Console.WriteLine(" After: a = {0}, b = {1}", a, b);
        Console.WriteLine("Before: c = {0}, d = {1}", c, d);
        Swap<double>(ref c, ref d);    // 실수형 변수로 호출
        Console.WriteLine(" After: c = {0}, d = {1}", c, d);
    }
}
```

실행 결과 :

```
Before: a = 1, b = 2
 After: a = 2, b = 1
Before: c = 1.5, d = 2.5
 After: c = 2.5, d = 1.5
```

Main 메소드 [1/2]

- C# 응용 프로그램의 시작점
- Main 메소드의 기본 형태

```
public static void Main(string[] args) {
        // ...
}
```

  - 매개변수 - 명령어 라인으로부터 스트링 전달
  - 명령어 라인으로부터 스트링 전달 방법

```
c:\>실행 파일명  인수1  인수2  ... 인수n
```

  - args[0] = 인수1, args[1] = 인수2, args[n-1] = 인수n

Main 메소드 [2/2]

[예제 4.9 - CommandLineArgsApp.cs]

```
using System;
class CommandLineArgsApp {
    public static void Main(string[] args) {
        for (int i = 0; i < args.Length; ++i)
            Console.WriteLine("Argument[{0}] = {1}", i, args[i]);
    }
}
```

실행 방법 :

C:₩> CommandLineArgsApp 12 Medusa 5.26

실행 결과 :

Argument[0] = 12
Argument[1] = Medusa
Argument[2] = 5.26

메소드 중복 [1/2]

- 시그네처(signature)
  - 메소드를 구분하는 정보
    - 메소드 이름
    - 매개변수의 개수
    - 매개변수의 자료형
    - 메소드 반환형 제외
- 메소드 중복(method overloading)
  - 메소드의 이름은 같은데 매개변수의 개수와 형이 다른 경우
  - 호출시 컴파일러에 의해 메소드 구별
- 메소드 중복 예

```
void SameNameMethod(int i) { /* ... */ }        // 첫 번째 형태
void SameNameMethod(int i, int j) { /* ... */ } // 두 번째 형태
```

메소드 중복 [2/2]

[예제 4.10 - MethodOverloadingApp.cs]

```
using System;
class MethodOverloadingApp {
    void SomeThing() {
        Console.WriteLine("SomeThing() is called.");
    }
    void SomeThing(int i) {
        Console.WriteLine("SomeThing(int) is called.");
    }
    void SomeThing(int i, int j) {
        Console.WriteLine("SomeThing(int,int) is called.");
    }
    void SomeThing(double d) {
        Console.WriteLine("SomeThing(double) is called.");
    }
    public static void Main() {
        MethodOverloadingApp obj = new MethodOverloadingApp();
        obj.SomeThing();                obj.SomeThing(526);
        obj.SomeThing(54, 526);         obj.SomeThing(5.26);
    }
}
```

실행 결과 :

```
SomeThing() is called.
SomeThing(int) is called.
SomeThing(int,int) is called.
SomeThing(double) is called.
```

생성자

- 생성자(constructor)
  - 객체가 생성될 때 자동으로 호출되는 메소드
  - 클래스 이름과 동일하며 반환형을 갖지 않음
  - 주로 객체를 초기화하는 작업에 사용
  - 생성자 중복 가능 – 예제 4.11
- 예)

```
class Fraction {
    // ....
    public Fraction(int a, int b) {        // 생성자
        numerator = a;
        denominator = b;
    }
}
// ...
Fraction f = new Fraction(1, 2);
```

정적 생성자 [1/2]

- 정적 생성자(static constructor)
  - 수정자가 static으로 선언된 생성자
  - 매개변수와 접근 수정자를 가질 수 없음
  - 클래스의 정적 필드를 초기화할 때 사용
  - Main() 메소드보다 먼저 실행

- 정적 필드 초기화 방법
  - 정적 필드 선언과 동시에 초기화
  - 정적 생성자 이용

정적 생성자 [2/2]

[예제 4.12 - StaticConstructorApp.cs]

```
using System;
class StaticConstructor {
    static int staticWithInitializer = 100;
    static int staticWithNoInitializer;
    static Constructor() {                    // 매개변수와 접근 수정자를 가질 수 없다.
        staticWithNoInitializer = staticWithInitializer + 100;
    }
    public static void PrintStaticVariable() {
        Console.WriteLine("field 1 = {0}, field 2 = {1}",
        staticWithInitializer, staticWithNoInitializer);
    }
}

class StaticConstructorApp {
    public static void Main() {
        StaticConstructor.PrintStaticVariable();
    }
}
```

실행 결과 :

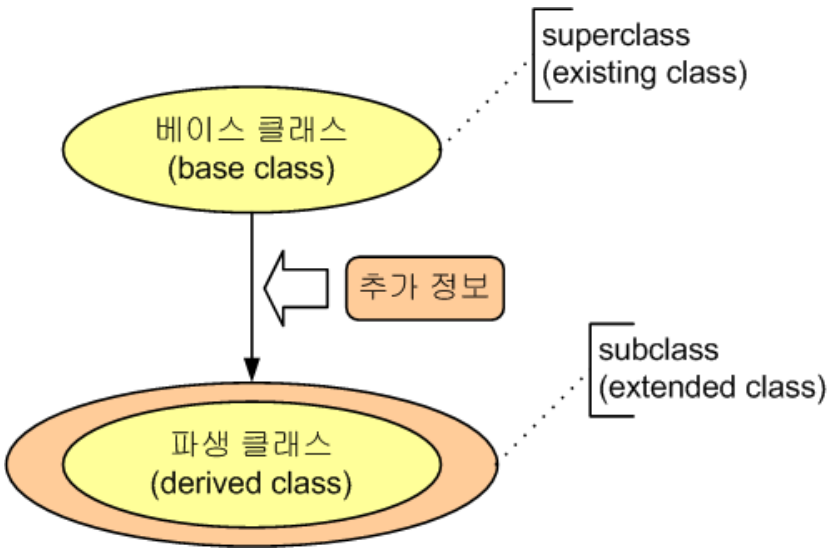
```
field 1 = 100, field 2 = 200
```

소멸자

- 소멸자(destructor) – 예제 4.13
  - 클래스의 객체가 소멸될 때 필요한 행위를 기술한 메소드
  - 소멸자의 이름은 생성자와 동일하나 이름 앞에 ~(tilde)를 붙임
- Finalize() 메소드
  - 컴파일 시 소멸자를 Finalize() 메소드로 변환해서 컴파일
  - Finalize() 메소드 재정의할 수 없음
  - 객체가 더 이상 참조되지 않을때 GC(Garbage Collection)에 의해 호출
- Dispose() 메소드 – 예제 4.14
  - CLR에서 관리되지 않은 자원을 직접 해제할 때 사용
  - 자원이 스코프를 벗어나면 즉시 시스템에 의해 호출

파생 클래스 [1/3]

- 파생 클래스 개념



- 상속(inheritance)
  - 베이스 클래스의 모든 멤버들이 파생 클래스로 전달 되는 기능
  - 클래스의 재사용성(reusability) 증가
- 상속의 종류
  - 단일 상속
    - 베이스 클래스 1개
  - 다중 상속
    - 베이스 클래스 1개 이상
- C#은 단일 상속만 지원

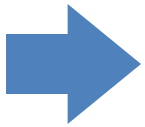
파생 클래스 [2/3]

- 파생 클래스의 정의 형태

```
[class-modifiers] class DerivedClassName : BaseClassName {
        // member declarations
}
```

- 파생 클래스 예

```
class BaseClass {
        int a;
        void MethodA{
                //...
        }
}
```



```
class DerivedClass : BaseClass {
        int b;
        void MethodB{
                //...
        }
}
```

파생 클래스 [3/3]

- 파생 클래스의 필드
  - 클래스의 필드 선언 방법과 동일
  - 베이스 클래스의 필드명과 다른 경우 - 상속됨
  - 베이스 클래스의 필드명과 동일한 경우 - 숨겨짐
    - base 지정어 – 베이스 클래스 멤버 참조 - 예제 4.15
- 파생 클래스의 생성자
  - 형태와 의미는 클래스의 생성자와 동일
  - 명시적으로 호출하지 않으면, 기본 생성자가 컴파일러에 의해 자동적으로 호출 – 예제 4.16
  - base()
    - 베이스 클래스의 생성자를 명시적으로 호출 – 예제 4.17
  - 실행과정
    - 필드의 초기화 부분 실행
    - 베이스 클래스의 생성자 실행
    - 파생 클래스의 생성자 실행

메소드 재정의

- 메소드 재정의(method overriding)
  - 베이스 클래스에서 구현된 메소드를 파생 클래스에서 구현된 메소드로 대체
  - 메소드의 시그네처가 동일한 경우 – 메소드 재정의
  - 메소드의 시그네처가 다른 경우 – 메소드 중복
  - 예제 4.18

가상 메소드 / 봉인 메소드

- 가상 메소드(virtual method)
  - 지정어 virtual로 선언된 인스턴스 메소드
  - 파생 클래스에서 재정의해서 사용할 것임을 알려주는 역할
    - new 지정어 – 객체 형에 따라 호출
    - override 지정어 – 객체 참조가 가리키는 객체에 따라 호출
- 봉인 메소드(sealed method)
  - 수정자가 sealed로 선언된 메소드
  - 파생 클래스에서 재정의를 허용하지 않음
  - 봉인 클래스 – 모든 메소드는 묵시적으로 봉인 메소드

추상 클래스 [1/2]

- 추상 클래스(abstract class)
  - 추상 메소드를 갖는 클래스
    - 추상 메소드(abstract method)
      - 실질적인 구현을 갖지 않고 메소드 선언만 있는 경우

- 추상 클래스 선언 방법

```
abstract class AbstractClass {
    public abstract void MethodA();
    void MethodB() {
        // ...
    }
}
```

추상 클래스 [2/2]

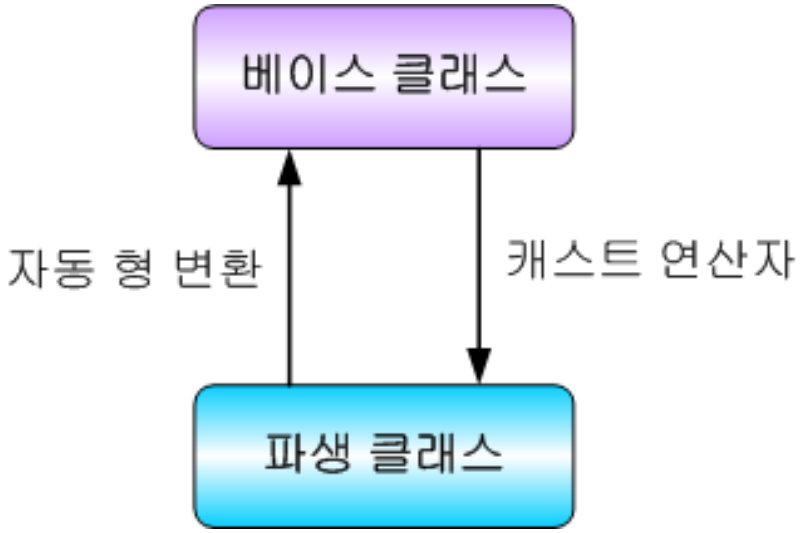
- 구현되지 않고, 단지 외형만을 제공
  - 추상 클래스는 객체를 가질 수 없음
  - 다른 외부 클래스에서 메소드를 일관성 있게 다루기 위한 방법 제공
  - 다른 클래스에 의해 상속 후 사용 가능

- abstract 수정자는 virtual 수정자의 의미 포함
  - 추상 클래스를 파생 클래스에서 구현
    - override 수정자를 사용하여 추상 메소드를 재정의
    - 접근 수정자 항상 일치

메소드 설계

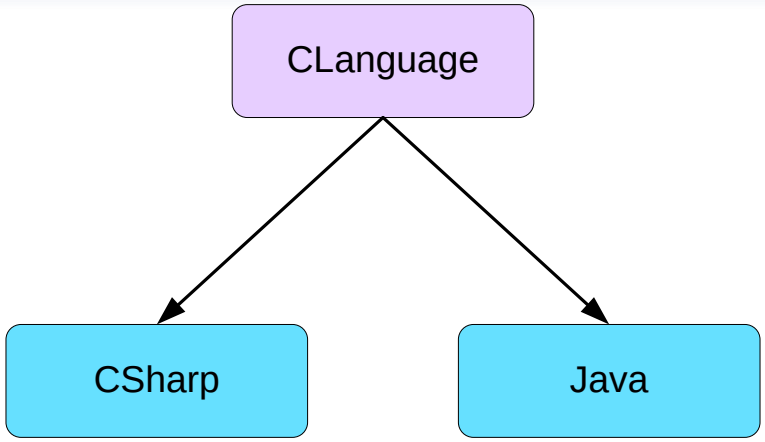
- 메소드를 파생 클래스에서 재정의하여 사용
  - C# 프로그래밍에 유용한 기능
  - 베이스 클래스에 있는 메소드에 작업을 추가하여 새로운 기능을 갖는 메소드 정의 – base 지정어 사용
  - 예제 4.21 – 파생 클래스에서 기능을 추가하여 재정의한 프로그램 예제

클래스형 변환 [1/2]



- 상향식 캐스트 (캐스팅-업) – 타당한 변환
  - 파생 클래스형의 객체가 베이스 클래스형의 객체로 변환
- 하향식 캐스트 (캐스팅-다운) – 타당하지 않은 변환
  - 캐스트 연산자 사용 – 예외 발생

클래스형 변환 [2/2]




```
void Dummy(CLanguage obj) {
    // ...
}
// ...
CSharp cs = new CSharp();
Dummy(cs);        // OK
```

```
void Dummy(CSharp obj) {
    // ...
}
// ...
CLanguage c = new CLanguage();
Dummy(c);        // 에러
```

dummy((CSharp)c) // 예외 발생

다형성

- 다형성(polymorphism)
  - 적용하는 객체에 따라 메소드의 의미가 달라지는 것
  - C# 프로그래밍 – virtual 과 override의 조합으로 메소드 선언
    – 예제 4.24

```
CLanguage c = new Java();
c.Print();
```

c의 형은 CLanguage이지만
Java 클래스의 객체를 가리킴

구조체 [1/2]

- 구조체(struct)
  - 클래스와 동일하게 객체의 구조와 행위를 정의하는 방법
  - 클래스 – 참조형, 구조체 – 값형
  - 예제 4.25 – 구조체를 선언하여 활용한 예제
- 구조체의 형태

```
[struct-modifiers] struct StructName {
        // member declarations
}
```

- 구조체의 수정자
  - public, protected, internal, private, new

구조체 [2/2]

- 구조체와 클래스 차이점

① 클래스는 참조형이고 구조체는 값형이다.

② 클래스 객체는 힙에 저장되고 구조체 객체는 스택에 저장된다.

③ 배정 연산에서 클래스는 참조가 복사되고 구조체는 내용이 복사된다.

④ 구조체는 상속이 불가능하다.

⑤ 구조체는 소멸자를 가질 수 없다.

⑥ 구조체의 멤버는 초기값을 가질 수 없다.

네임스페이스

- 네임스페이스(namespace)
  - 서로 연관된 클래스나 인터페이스, 구조체, 열거형, 델리게이트, 하위 네임스페이스를 하나의 단위로 묶어주는 것
  - 예)
    - 여러 개의 클래스와 인터페이스, 구조체, 열거형, 델리게이트 등을 하나의 그룹으로 다루는 수단을 제공
    - 클래스의 이름을 지정할 때 발생 되는 이름 충돌 문제 해결
- 네임스페이스 선언

```
namespace NamespaceName {
      // 네임스페이스에 포함할 항목을 정의
}
```

- 네임스페이스 사용

```
using NamespaceName; // 사용하고자하는 네임스페이스 명시
```