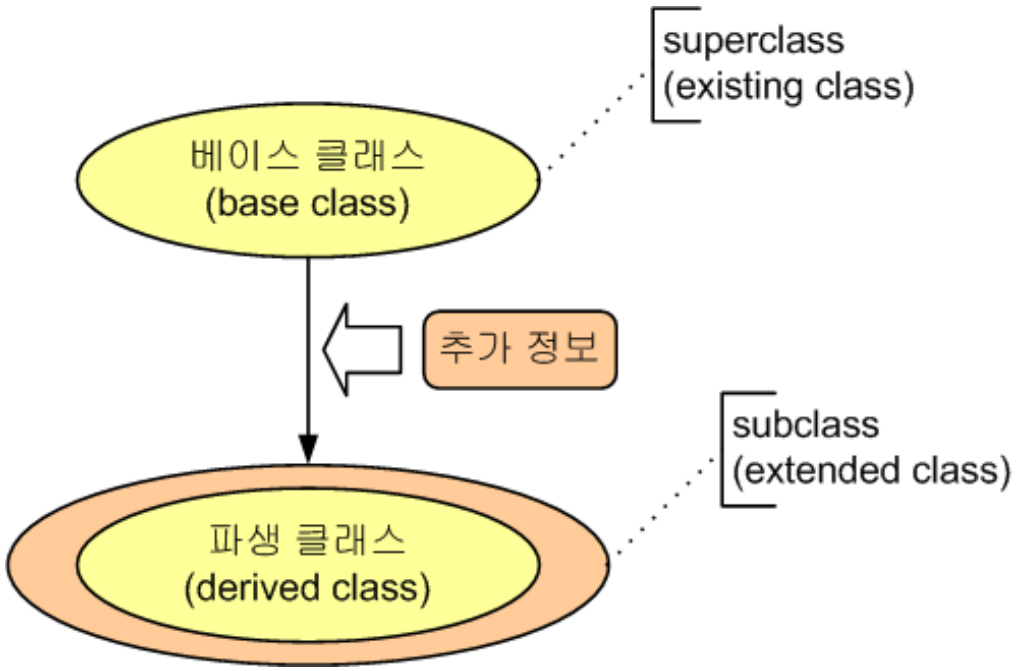
객체지향 프로그래밍

C# - derived Class, Interface, Generic, Exception

KYUNGSUNG UNIVERSITY SINCE 1955

파생클래스(Derived Class)

- 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

파생클래스(Derived Class)

- 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{
                //...
        }
}
```

파생클래스(Derived Class)

- 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 HiddenNameApp
- 파생 클래스의 생성자
  - 형태와 의미는 클래스의 생성자와 동일
  - 명시적으로 호출하지 않으면, 기본 생성자가 컴파일러에 의해 자동적으로 호출 – 예제 DerivedConstructorApp
  - base()
    - 베이스 클래스의 생성자를 명시적으로 호출 – 예제 BaseCallApp
  - 실행과정
    - 필드의 초기화 부분 실행
    - 베이스 클래스의 생성자 실행
    - 파생 클래스의 생성자 실행

```
using System;

namespace HiddenNameApp
{
    class BaseClass
    {
        protected int a = 1;
        protected int b = 2;
    }
    class DerivedClass : BaseClass
    {
        new int a = 3;
        new double b = 4.5;
        public void Output()
        {
            Console.WriteLine("BaseClass : a={0}, DerivedClass:a={1}", base.a, a);
            Console.WriteLine("BaseClass : b={0}, DerivedClass:b={1}", base.b, b);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DerivedClass obj = new DerivedClass();
            obj.Output();
        }
    }
}
```

```
using System;

namespace DerivedConstructorApp
{
    class BaseClass
    {
        public BaseClass()
        {
            Console.WriteLine("BaseClass Constructor ...");
        }
    }
    class DerivedClass : BaseClass
    {
        public DerivedClass()
        {
            Console.WriteLine("DerivedClass Constructor ...");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DerivedClass obj = new DerivedClass();
            Console.WriteLine("In the main ...");
        }
    }
}
```

```
using System;

namespace BaseCallApp
{
    class BaseClass
    {
        public int a, b;
        public BaseClass()
        {
            a = 1; b = 1;
        }
        public BaseClass(int a, int b)
        {
            this.a = a; this.b = b;
        }
    }
    class DerivedClass : BaseClass
    {
        public int c;
        public DerivedClass()
        {
            c = 1;
        }
        public DerivedClass(int a, int b, int c)
            : base(a, b)
        {
            this.c = c;
        }
    }
```

```
class Program
    {
        static void Main(string[] args)
        {
            DerivedClass obj1 = new DerivedClass();
            DerivedClass obj2 = new DerivedClass(1, 2, 3);
            Console.WriteLine("a = {0}, b = {1}, c = {2}",
                obj1.a, obj1.b, obj1.c);
            Console.WriteLine("a = {0}, b = {1}, c = {2}",
                obj2.a, obj2.b, obj2.c);
        }
    }
}
```

메소드 재정의(Method overriding)

- 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 OverridingAndOverloadingApp

```
using System;

namespace OverridingAndOverloadingApp
{
    class BaseClass
    {
        public void MethodA()
        {
            Console.WriteLine("In the BaseClass ...");
        }
    }
    class DerivedClass : BaseClass
    {
        new public void MethodA()
        {               // Overriding
            Console.WriteLine("In DerivedClass ... Overriding !!!");
        }
        public void MethodA(int i)
        {            // Overloading
            Console.WriteLine("In DerivedClass ... Overloading !!!");
        }
    }
```

```
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass obj1 = new BaseClass();
            DerivedClass obj2 = new DerivedClass();
            obj1.MethodA();
            obj2.MethodA();
            obj2.MethodA(1);
        }
    }
}
```

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

```
using System;

namespace VirtualMethodApp
{
    class BaseClass
    {
        virtual public void MethodA()
        {
            Console.WriteLine("Base MethodA");
        }
        virtual public void MethodB()
        {
            Console.WriteLine("Base MethodB");
        }
        virtual public void MethodC()
        {
            Console.WriteLine("Base MethodC");
        }
    }
```

```
class DerivedClass : BaseClass
    {
        new public void MethodA()
        {
            Console.WriteLine("Derived MethodA");
        }
        override public void MethodB()
        {
            Console.WriteLine("Derived MethodB");
        }
        public void MethodC()
        {
            Console.WriteLine("Derived MethodC");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass s = new DerivedClass();
            s.MethodA();
            s.MethodB();
            s.MethodC();
        }
    }
}
```

추상 클래스(Abstract Class)

- 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(Abstract Class)

- 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

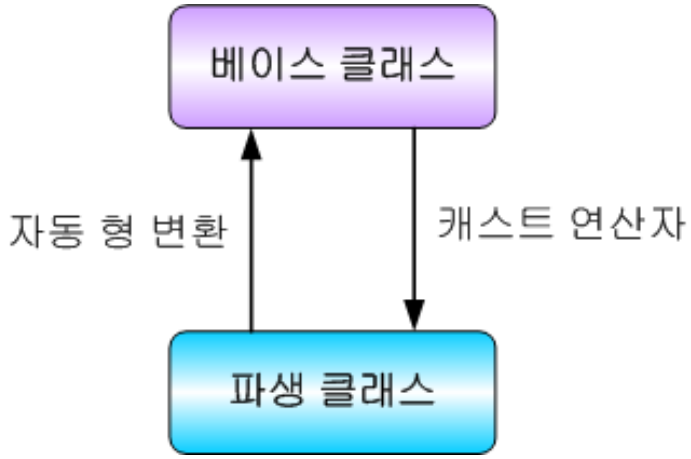
```
using System;

namespace AbstractClassApp
{
    abstract class AbstractClass
    {
        public abstract void MethodA();
        public void MethodB()
        {
            Console.WriteLine("Implementation of MethodB()");
        }
    }
    class ImpClass : AbstractClass
    {
        override public void MethodA()
        {
            Console.WriteLine("Implementation of MethodA()");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            ImpClass obj = new ImpClass();
            obj.MethodA();
            obj.MethodB();
        }
    }
}
```

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 AddendumApp – 파생 클래스에서 기능을 추가하여 재정의한 프로그램 예제

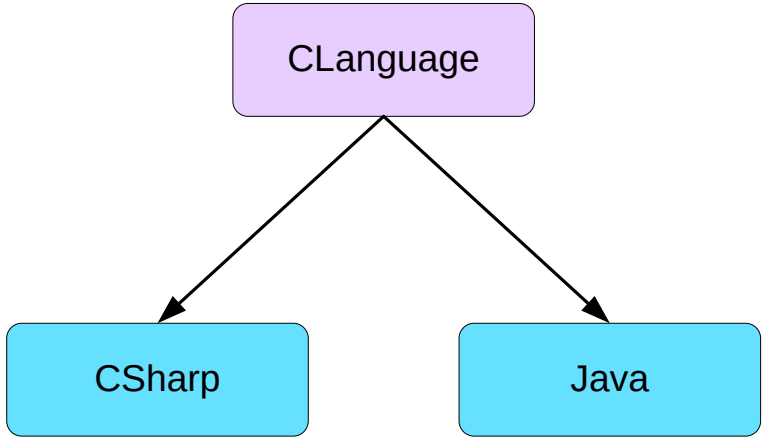
```
using System;

namespace AddendumApp
{
    class BaseClass
    {
        public void MethodA()
        {
            Console.WriteLine("do BaseClass Task.");
        }
    }
    class DerivedClass : BaseClass
    {
        new public void MethodA()
        {
            base.MethodA();
            Console.WriteLine("do DerivedClass Task.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            DerivedClass obj = new DerivedClass();
            obj.MethodA();
        }
    }
}
```

클래스 형 변환



- 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

클래스 형 변환




```
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

is : 특정 객체와의 타입과 호환이 가능한지 확인하는 연산자 입니다.

```
using System;

namespace ClassTypeConversionApp
{
    class CLanguage
    {
        public bool Equal(object obj)
        {
            if (obj is CLanguage)
               return true;
            else
               return false;
        }
    }
    class CSharp : CLanguage
    {
        new public bool Equal(object obj)
        {
            return (obj is CSharp) ? true : false;
        }
    }
```

```
class Program
   {
       static void Main(string[] args)
       {
           CLanguage c = new CLanguage();
           CSharp cs = new CSharp();

           if (c.Equal(cs))
              Console.WriteLine("casting up is valid.");
           else
              Console.WriteLine("casting up is not valid.");
           if (cs.Equal(c))
              Console.WriteLine("casting down is valid.");
           else
              Console.WriteLine("casting down is not valid.");
       }
   }
}
```

다형성

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

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

c의 형은 CLanguage이지만
Java 클래스의 객체를 가리킴

```
using System;

namespace VirtualAndOverrideApp
{
    class CLanguage
    {
        virtual public void Print()
        {
            Console.WriteLine("In the Clanguage class ...");
        }
    }
    class Java : CLanguage
    {
        override public void Print()
        {
            Console.WriteLine("In the Java class ...");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            CLanguage c = new Java();
            c.Print();
        }
    }
}
```

```
using System;

namespace PolymorphismApp
{
    class BaseClass
    {
        public virtual void Output()
        {
            Console.WriteLine("In the Base class ...");
        }
    }
    class DerivedClass : BaseClass
    {
        public override void Output()
        {
            Console.WriteLine("In the Derived class ...");
        }
    }
```

```
 class Program
    {
        static void Print(BaseClass obj)
        {
            obj.Output();
        }
        static void Main(string[] args)
        {
            BaseClass obj1 = new BaseClass();
            DerivedClass obj2 = new DerivedClass();
            Print(obj1);
            Print(obj2);
        }
    }
}
```

인터페이스

- 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

- 인터페이스 선언 형태

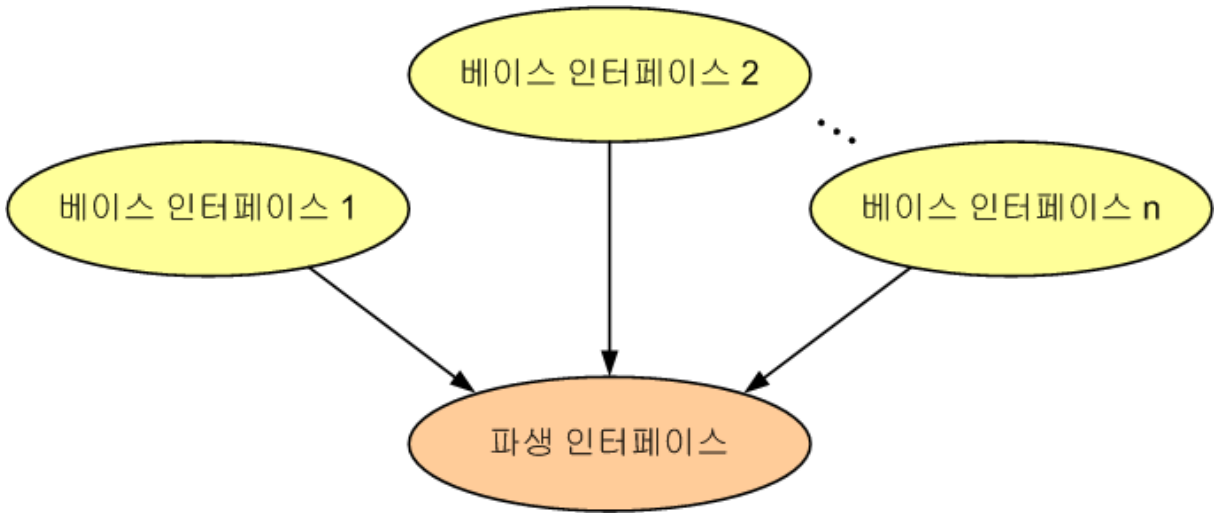
```
[interface-modifiers] [partial] interface InterfaceName {
        // interface body
}
```

- 인터페이스 확장 형태

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

인터페이스

- 인터페이스의 다중 상속



인터페이스의 구현

- 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
}
```

```
using System;

namespace ImplementingInterfaceApp
{
    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);
        }
    }
```

```
    class Program
    {
        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);
        }
    }
}
```

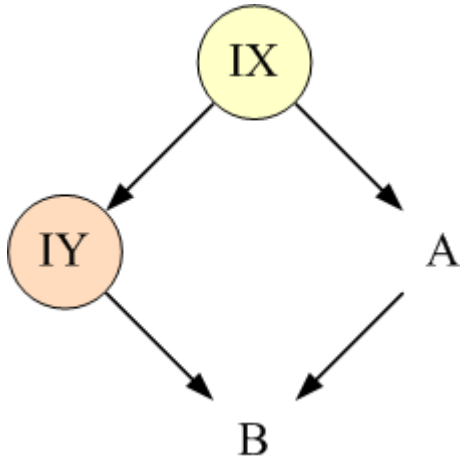
인터페이스의 구현

- 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 { }
```



```
using System;

namespace DiamondApp
{
    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 Program
    {
        public static void Main()
        {
            B obj = new B();
            obj.XMethod(5); obj.YMethod(10);
            Console.WriteLine("a = {0}, b = {1}",
                obj.PropertyA, obj.PropertyB);
        }
    }
}
```

인터페이스와 추상클래스

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

```
using System;

namespace StructImpApp
{
    interface IVector
    {
        void Insert(int n);     // 배열에 원소를 삽입한다.
        void ScalarSum(int n); // 배열에 각 원소에 스칼라 값을 더한다.
        void PrintVector();     // 배열에 있는 모든 원소를 출력한다.
    }
```

```
    struct Vector : IVector
    {
        private int[] v;
        private int index, size;
        public Vector(int size)
        {  // »ý¼ºÀÚ
            v = new int[size];
            this.size = size;
            index = 0;
        }
        public void Insert(int n)
        {
            if (index >= size)
                Console.WriteLine("Error : array overflow");
            else v[index++] = n;
        }
        public void ScalarSum(int n)
        {
            for (int i = 0; i < index; i++) v[i] += n;
        }
        public void PrintVector()
        {
            Console.Write("Contents:");
            for (int i = 0; i < index; i++)
                Console.Write(" " + v[i]);
            Console.WriteLine();
        }
    }
```

```
class Program
    {
        public static void Main()
        {
            Vector a = new Vector(100);
            int n;
            while (true)
            {           // 0이 입력될 때까지 반복한다.
                n = Console.Read() - '0';
                if (n == 0) break;
                a.Insert(n);
            }
            a.PrintVector();
            a.ScalarSum(10);
            a.PrintVector();
        }
    }
}
```

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
```

```
using System;

namespace Shape
{
    public class Rectangle
    {
        public int height;
        public int width;
        override public string ToString()
        {
            return string.Format("Width: {0}, Height: {1}", width, height);
        }
    }
}
```

```
using System;
using Shape;

namespace NamespaceApp
{
    class Program
    {
        static void Main(string[] args)
        {
            Rectangle rect = new Rectangle();
            rect.width = 10;
            rect.height = 20;
            Console.WriteLine("rect : " + rect);
        }
    }
}
```

제네릭 클래스

- 형 매개변수(type parameter)를 가지는 클래스
  - 형 매개변수를 이용하여 필드나 지역변수에 사용
  - 실제 형 정보는 객체 생성 시에 전달받음

- 정의 형태

```
[modifiers] class ClassName<TypeParameters> {
            // ... class body
}
```

- 형식 매개변수 형(formal parameter type)
  - 제네릭 클래스를 선언할 때 사용한 형 매개변수
  - class SimpleGeneric<T> { ... }

- 실제 형 인자(actual type argument)
  - 제네릭 클래스에 대한 객체를 생성할 때 주는 자료형
  - new SimpleGeneric<Int32>(10);

```
using System;
namespace GenericClassApp
{
    class SimpleGeneric<T>
    {
        private T[] values;
        private int index;

        public SimpleGeneric(int len)
        { // Constructor
            values = new T[len];
            index = 0;
        }

        public void add(params T[] args)
        {
            foreach (T e in args)
                values[index++] = e;
        }

        public void print()
        {
            foreach (T e in values)
                Console.Write(e + " ");
            Console.WriteLine();
        }
    }
```

```
    public class Program
    {
        public static void Main()
        {
            SimpleGeneric<Int32> gInteger = new SimpleGeneric<Int32>(10);
            SimpleGeneric<Double> gDouble = new SimpleGeneric<Double>(10);

            gInteger.add(1, 2);
            gInteger.add(1, 2, 3, 4, 5, 6, 7);
            gInteger.add(0);
            gInteger.print();

            gDouble.add(10.0, 20.0, 30.0);
            gDouble.print();
        }
    }
}
```

제네릭 인터페이스

- 형 매개변수를 가지는 인터페이스
  - 형 매개변수의 선언 외에 일반 인터페이스를 구현하는 과정과 동일

- 정의 형태

```
[modifiers] interface IName<TypeParameters> {
            // ... interface body
}
```

```
using System;

namespace GenericInterfaceApp
{
    interface IGenericInterface<T>
    {
        void setValue(T x);
        string getValueType();
    }

    class GenericClass<T> : IGenericInterface<T> {
        private T value;
        public void setValue(T x) {
            value = x;
        }

        public string getValueType() {
            return value.GetType().ToString();
        }
    }
```

```
    public class Program
    {
        public static void Main()
        {
            GenericClass<Int32> gInteger = new GenericClass<Int32>();
            GenericClass<String> gString = new GenericClass<String>();

            gInteger.setValue(10);
            gString.setValue("Text");

            Console.WriteLine(gInteger.getValueType());
            Console.WriteLine(gString.getValueType());
        }
    }
}
```

제네릭 메소드

- 제네릭 메소드(generic method)
  - 형 매개변수(type parameter)를 갖는 메소드

- 제네릭 메소드 정의 예

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

- 제네릭 메소드 호출 예

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

제네릭 메소드

- 형 매개변수의 중첩
  - 제네릭 메소드의 형 매개변수의 이름과 제네릭 클래스의 형 매개변수 이름이 같은 경우
  - 서로 독립된 형 매개변수의 개념을 가짐
    - 제네릭 클래스는 객체 생성 시에 형 매개변수를 전달받음
    - 제네릭 메소드는 호출 시에 유추하여 형 매개변수가 결정됨

```
using System;

namespace GenericMethodApp
{
    class Program
    {
        static void Swap<DataType>(ref DataType x, ref DataType y)
        {
            DataType temp;
              temp = x;
              x = y;
              y = temp;
        }
        static void Main(string[] args)
        {
            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);
        }
    }
}
```

형 매개변수의 제한

- 형 매개 변수의 범위
  - 프로그램의 유연성 ⇔ 신뢰성
- 프로그램의 신뢰성을 증진하기 위해 제네릭에 전달 가능한 자료형의 범위를 제한할 필요가 있음
- 제네릭 클래스를 작성시 한정
  - where 키워드 사용

<T> where T : S - S 형의 서브클래스형으로 제한한다.

형 매개변수의 제한

- 형 매개변수 사용시 한정

제한 조건	설명
where T : struct	T는 값형이어야 함
where T : class	T는 참조형이어야 함
where T : new()	T는 매개변수가 없는 생성자가 있어야 함
where T : MyClass	T는 MyClass의 파생 클래스이어야 함
where T : IMyInterface	T는 IMyInterface를 구현한 클래스이어야 함
where T : U	T는 U로부터 파생된 클래스이어야 함

```
using System;

namespace BoundedGenericApp
{
    class GenericType<T> where T : SystemException
    {
        private T value;

        public GenericType(T v)
        {
            value = v;
        }

        override public String ToString()
        {
            return "Type is " + value.GetType();
        }
    }
```

```
 public class Program
    {
        public static void Main()
        {
            GenericType<NullReferenceException> gNullEx = new GenericType<NullReferenceException>(new NullReferenceException());
            GenericType<IndexOutOfRangeException> gIndexEx = new GenericType<IndexOutOfRangeException>(new
IndexOutOfRangeException());
            // GenericType<String> gString = new GenericType<String>("Error"); //에러

            Console.WriteLine(gNullEx.ToString());
            Console.WriteLine(gIndexEx.ToString());
        }
    }
}
```

애트리뷰트

- 애트리뷰트의 특징
  - 어셈블리나 클래스, 필드, 메소드, 프로퍼티 등에 다양한 종류의 속성 정보를 추가하기 위해서 사용
  - 어셈블리에 메타데이터(metadata) 형식으로 저장되며, 이를 참조하는 .NET 프레임워크나 C# 또는 다른 언어의 컴파일러에 의해 다양한 용도로 사용

- 애트리뷰트의 정의 형태

```
[attribute AttributeName ("positional_parameter", named_parameter = value,...)]
```

- 애트리뷰트의 종류
  - 표준 애트리뷰트(.NET 프레임워크에서 제공)
  - 사용자 정의 애트리뷰트

표준 애트리뷰트

- Conditional 애트리뷰트
  - 조건부 메소드를 작성할 때 사용
  - C/C++ 언어에서 사용했던 전처리기 지시어를 이용하여 명칭을 정의 (#define)
  - System.Diagnostics를 사용해야 함

- Obsolete 애트리뷰트
  - 앞으로 사용되지 않을 메소드를 표시하기 위해서 사용
  - 해당 애트리뷰트를 가진 메소드를 호출할 경우 컴파일러는 컴파일 과정에서 애트리뷰트에 설정한 내용이 출력하는 경고를 발생

```
#define CSHARP
using System;
using System.Diagnostics;
namespace ConditionalAttrApp
{
    class ConditionalAttr
    {
        [Conditional("CSHARP")]
        public static void CsharpMethod()
        {
            Console.WriteLine("In the CSharp Method ...");
        }
        [Conditional("JAVA")]
        public static void JavaMethod()
        {
            Console.WriteLine("In the Java Method ...");
        }
    }

    class Program
    {
        public static void Main()
        {
            ConditionalAttr.CsharpMethod();
            ConditionalAttr.JavaMethod();
        }
    }
}
```

debug

```
using System;

namespace ObsoleteAttrApp
{
    class ObsoleteAttr
    {
        [Obsolete("경고, Obsolete Method입니다.")]
        public static void OldMethod()
        {
            Console.WriteLine("In the Old Method ...");
        }
        public static void NormalMethod()
        {
            Console.WriteLine("In the Normal Method ...");
        }
    }
    class Program
    {
        public static void Main()
        {
            ObsoleteAttr.OldMethod();
            ObsoleteAttr.NormalMethod();
        }
    }
}
```

사용자 정의 애트리뷰트

- 특징
  - System.Attribute 클래스에서 파생
    - 이름의 형태 : XxxxAttribute
  - 정의한 애트리뷰트를 사용할 때는 이름에서 Attribute가 제외된 부분만을 사용
  - 사용자 정의 애트리뷰트나 표준 애트리뷰트를 사용하기 위해서는 .NET 프레임워크가 제공하는 리플렉션 기능을 사용

```
// 사용자 정의 애트리뷰트를 정의한 예
public class AttributeNameAttribute: Attribute {
    // 생성자 정의
}
// ...
// 사용자 정의 애트리뷰트를 사용한 예
[AttributeName()]
```

```
using System;

namespace MyAttributeApp
{
    public class MyAttrAttribute : Attribute
    {   // 속성 클래스

        public MyAttrAttribute(string message)
        { // 생성자

            this.message = message;
        }
        private string message;
        public string Message
        {                   // 프로퍼티
            get { return message; }
        }
    }
```

```
    [MyAttr("This is Attribute test.")]
    class Program
    {
        public static void Main()
        {
            Type type = typeof(Program);
            object[] arr = type.GetCustomAttributes(typeof(MyAttrAttribute), true);
            if (arr.Length == 0)
                Console.WriteLine("This class has no custom attrs.");
            else
            {
                MyAttrAttribute ma = (MyAttrAttribute)arr[0];
                Console.WriteLine(ma.Message);
            }
        }
    }
}
```

예외

- 예외(exception)
  - 실행 시간에 발생하는 에러(run-time error)
  - 프로그램의 비정상적인 종료
  - 잘못된 실행 결과
  - 메소드의 호출과 실행, 부정확한 데이터, 그리고 시스템 에러 등 다양한 상황에 의해 야기
- 예외 처리(exception handling)
  - 기대되지 않은 상황에 대해 예외를 발생
  - 야기된 예외를 적절히 처리 (exception handler)
- 예외 처리를 위한 방법을 언어 시스템에서 제공
  - 응용프로그램의 신뢰성(reliability) 향상
  - 예외 검사와 처리를 위한 프로그램 코드를 소스에 깔끔하게 삽입

예외 정의

- 예외도 하나의 객체로 취급
  - 따라서, 먼저 예외를 위한 클래스를 정의하여야 함
- 예외 클래스
  - 모든 예외는 형(type)이 Exception 클래스 또는 그의 파생 클래스들 중에 하나로부터 확장된 클래스의 객체
  - 일반적으로 프로그래머는 Exception 클래스의 파생 클래스인 ApplicationException 클래스를 확장하여 새로운 예외 클래스를 정의하여 사용
- 예외를 명시적으로 발생시키면 예외를 처리하는 예외 처리기가 반드시 필요함
- 예외에 관련된 메시지를 스트링 형태로 예외 객체에 담아 전달 가능

```
using System;

namespace UserExceptionApp
{
    class UserErrException : ApplicationException
    {
        public UserErrException(string s) : base(s) { }
    }

    class Program
    {
        public static void Main()
        {
            try
            {
                throw new UserErrException("throw a exception with a message");
            }
            catch (UserErrException e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

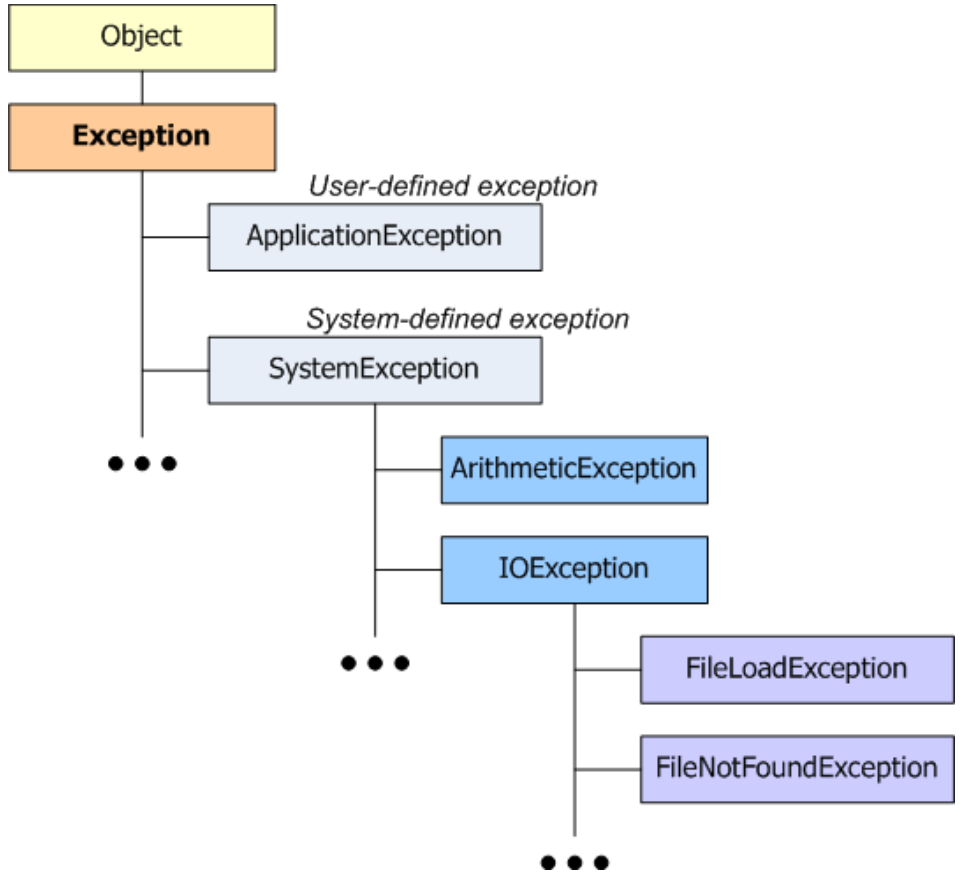
예외 정의

- 시스템 정의 예외 (system-defined exception)
  - 프로그램의 부당한 실행으로 인하여 시스템에 의해 묵시적으로 일어나는 예외
  - SystemException클래스나 IOException 클래스로부터 확장된 예외
  - CLR에 의해 자동적으로 생성
  - 야기된 예외에 대한 예외 처리기의 유무를 컴파일러가 검사하지 않음(unchecked exception)

- 프로그래머 정의 예외 (programmer-defined exception)
  - 프로그래머에 의해 의도적으로 야기되는 프로그래머 정의 예외
  - 프로그래머 정의 예외는 발생한 예외에 대한 예외 처리기가 존재하는지 컴파일러에 의해 검사 (checked exception)

예외 정의

- 예외 클래스의 계층도



예외 정의

- 시스템 정의 예외의 종류
  - ArithmeticException
    - 산술 연산 시 발생하는 예외
  - IndexOutOfRangeException
    - 배열, 스트링, 벡터 등과 같이 인덱스를 사용하는 객체에서 인덱스의 범위가 벗어날 때 발생하는 예외
  - ArrayTypeMismatchException
    - 배열의 원소에 잘못된 형의 객체를 배정하였을 때 발생하는 예외
  - InvalidCastException
    - 명시적 형 변환이 실패할 때 발생하는 예외
  - NullReferenceException
    - null을 사용하여 객체를 참조할 때 발생하는 예외
  - OutOfMemoryException
    - 메모리 할당(new)이 실패하였을 때 발생하는 예외

```
using System;

namespace DivByZeroExceptionApp
{
    class Program
    {
        public static void Main()
        {
            int i = 1, j = 0, k;
            k = i / j;
        }
    }
}
```

```
using System;

namespace IndexOutOfRangeApp
{
    class Program
    {
        public static void Main()
        {
            int[] vector = { 1, 2, 3 };
            vector[4] = 4;
        }
    }
}
```

```
using System;

namespace DivByZeroExceptionApp
{
    class Program1
    {
        public static void Main()
        {
            Console.WriteLine("3개의 정수를 입력하세요 : ");
            int sum = 0, n = 0;
            for(int i = 0; i < 3; i++)
            {
                Console.Write( i + " >> ");
                n = Convert.ToInt32(Console.ReadLine());
                sum += n;
            }
            Console.WriteLine("합은 "+ sum);
        }
    }
}
```

```
using System;

namespace DivByZeroExceptionApp
{
    class Program1
    {
        public static void Main()
        {
            Console.WriteLine("3개의 정수를 입력하세요 : ");
            int sum = 0, n = 0;
            for(int i = 0; i < 3; i++)
            {
                Console.Write( i + " >> ");
                try
                {
                    n = Convert.ToInt32(Console.ReadLine());
                } catch(FormatException e)
                {
                    Console.WriteLine("정수가 아닙니다. 다시 입력해 주세요");
                    i--;
                    continue;
                }
                sum += n;
            }
            Console.WriteLine("합은 "+ sum);
        }
    }
}
```

예외 발생

- 묵시적 예외 발생
  - 시스템 정의 예외로 CLR에 의해 발생
  - 시스템에 의해 발생되므로 프로그램 어디서나 발생 가능
  - 프로그래머가 처리하지 않으면 디폴트 예외 처리기(default exception handler)에 의해 처리

- 명시적 예외 발생
  - throw 문을 이용하여 프로그래머가 의도적으로 발생

```
throw ApplicationExceptionObject;
```

  - 프로그래머 정의 예외는 생성된 메소드 내부에 예외를 처리하는 코드 부분인 예외 처리기를 두어 직접 처리해야 함

```
using System;

namespace UserExThrowApp
{
    class UserException : ApplicationException { }
    class Program
    {
        static void Method()
        {
            throw new UserException();
        }
        public static void Main()
        {
            try
            {
                Console.WriteLine("Here: 1");
                Method();
                Console.WriteLine("Here: 2");
            }
            catch (UserException)
            {
                Console.WriteLine("User-defined Exception");
            }
        }
    }
}
```

```
using System;

namespace SystemExThrowApp
{
    class SystemExThrow
    {
        public static void Exp(int ptr)
        {
            if (ptr == 0)
                throw new NullReferenceException();
        }
    }
    class Program
    {
        public static void Main()
        {
            int i = 0;
            SystemExThrow.Exp(i);
        }
    }
}
```

```
using System;

namespace UserHandlerApp
{
    class UserExceptionOne : ApplicationException { }
    class UserExceptionTwo : ApplicationException { }
    class Program
    {
        static void Method(int i)
        {
            if (i == 1) throw new UserExceptionOne();
            else throw new UserExceptionTwo();
        }
```

```
 public static void Main()
        {
            try
            {
                Console.WriteLine("Here: 1");
                Method(2);
                Console.WriteLine("Here: 2");
            }
            catch (UserExceptionOne)
            {
                Console.WriteLine("UserExceptionOne is occurred!!!");
            }
            catch (UserExceptionTwo)
            {
                Console.WriteLine("UserExceptionTwo is occurred!!!");
            }
            Console.WriteLine("Here: 3");
        }
    }
}
```

```
using System;

namespace SystemHandlerApp
{
    class Program
    {
        public static void Main()
        {
            int[] vector = { 1, 2, 3 };
            try
            {
                Console.WriteLine("Here: 1");
                vector[4] = 4;
                Console.WriteLine("Here: 2");
            }
            catch (IndexOutOfRangeException)
            {
                Console.WriteLine("System Exception is occurred!!!");
            }
            Console.WriteLine("Here: 3");
        }
    }
}
```

예외 처리

- 에러 처리 구문(try-catch-finally 구문)
  - 예외를 검사하고 처리해주는 문장
  - 구문 형태

```
try {
    // ...          "try 블록"
} catch (ExceptionType identifier) {
    // ...          "catch 블록"
} catch (ExceptionType identifier) {
    // ...          "catch 블록"
} finally {
    // ...          "finally 블록"
}
```

  - try 블록 : 예외 검사
  - catch 블록 : 예외 처리
  - finally 블록 : 종결 작업, 예외 발생과 무관하게 반드시 실행

예외 처리

- 에러 처리기의 실행 순서
  - ① try 블록 내에서 예외가 검사되고 또는 명시적으로 예외가 발생하면,
  - ② 해당하는 catch 블록을 찾아 처리하고,
  - ③ 마지막으로 finally 블록을 실행한다.

- Default 예외 처리기
  - 시스템 정의 예외가 발생됐는데도 불구하고 프로그래머가 처리하지 않을 때 작동
  - 단순히 에러에 대한 메시지를 출력하고 프로그램을 종료하는 기능

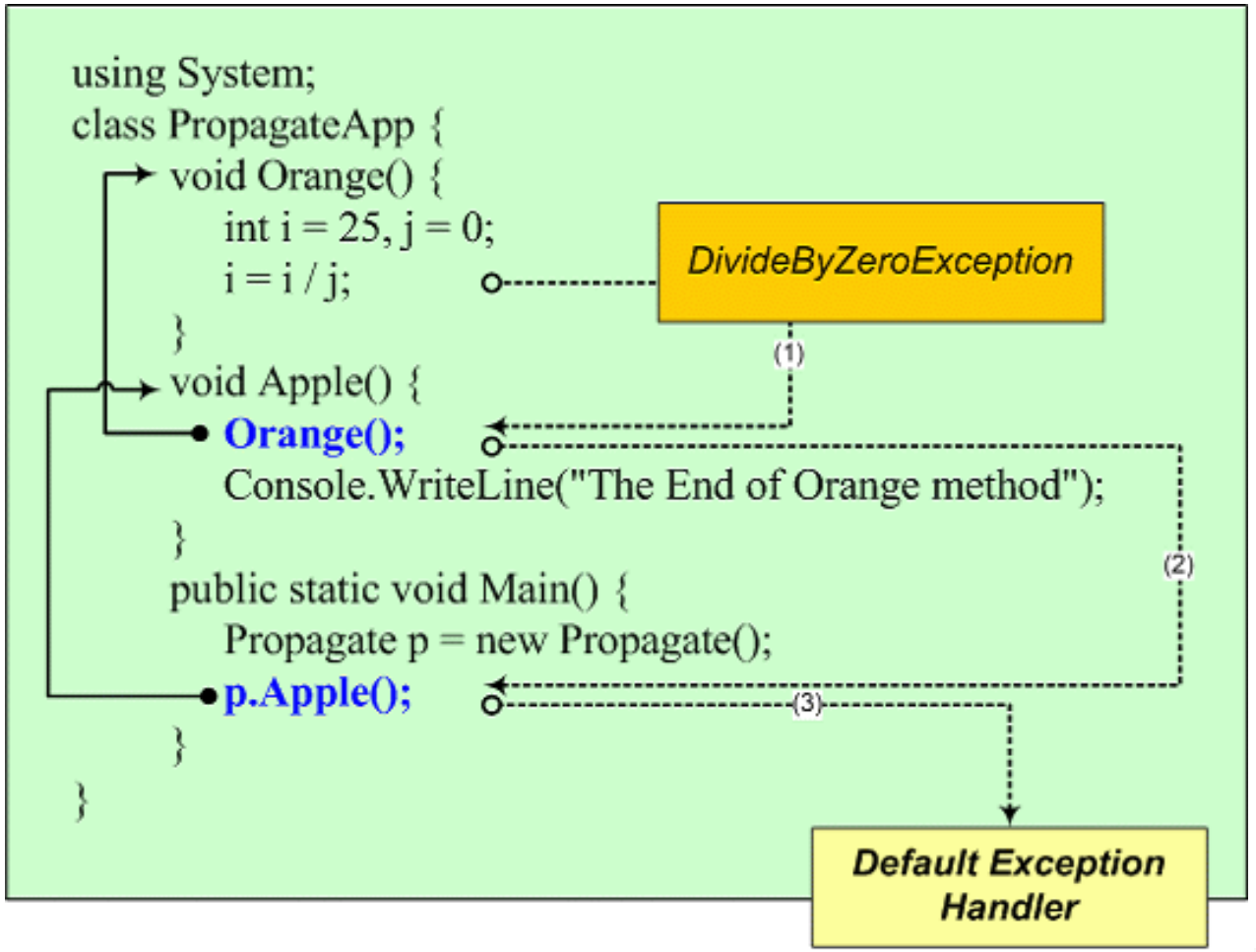
```
using System;

namespace FinallyClauseApp
{
    class Program
    {
        static int count = 0;
        public static void Main()
        {
            while (true)
            {
                try
                {
                    if (++count == 1) throw new Exception();
                    if (count == 3) break;
                    Console.WriteLine(count + ") No exception");
                }
                catch (Exception)
                {
                    Console.WriteLine(count + ") Exception thrown");
                }
                finally
                {
                    Console.WriteLine(count + ") in finally clause");
                }
            } // end while
            Console.WriteLine("Main program ends");
        }
    }
}
```

예외 전파

- 호출한 메소드로 예외를 전파(propagation)하여 특정 메소드에서 모아 처리
  - 예외 처리 코드의 분산을 막을 수 있음
- 예외 전파 순서
  - 예외를 처리하는 catch 블록이 없으면 호출한 메소드로 예외를 전파함
  - 예외처리기를 찾을 때까지의 모든 실행은 무시

예외 전파

- 예외 전파 과정


using System;
class PropagateApp {
void Orange() {
int i = 25, j = 0;
i = i / j;
}
void Apple() {
Orange();
Console.WriteLine("The End of Orange method");
}
public static void Main() {
Propagate p = new Propagate();
p.Apple();
}
}
DivideByZeroException
(1)
(2)
(3)
Default Exception Handler

```
using System;

namespace PropagateApp
{
    class Propagate
    {
        public void Orange()
        {
            int i = 25, j = 0;
            i = i / j;
            Console.WriteLine("End of Orange method");
        }
        public void Apple()
        {
            Orange();
            Console.WriteLine("End of Apple method");
        }
    }
```

```
    class Program
    {
        public static void Main()
        {
            Propagate p = new Propagate();
            try
            {
                p.Apple();
            }
            catch (ArithmeticException)
            {
                Console.WriteLine("ArithmeticException is processed");
            }
            Console.WriteLine("End of Main");
        }
    }
}
```

Reference

- ✓ C# 프로그래밍 입문, 오세만 외4, 생능출판
- ✓ 초보자를 위한 C# 200제, 강병익, 정보문화사
- ✓ 프랙티컬 C#, 이데이 히데유키, 김범준, 위키북스
- ✓ C#언어 프로그래밍 바이블, 김명렬 외1, 홍릉과학출판사
- ✓ C# and the .NET Platform, Andrew Troelsen, 장시혁, 사이텍미디어
- ✓ https://docs.microsoft.com/ko-kr/learn/browse/?products=dotnet&terms=c%23