# HCI 프로그래밍

## 04. 객체 지향 개념 & 클래스와 객체

HCI
Human Computer Interaction

# C 프로그램

```
struct _person {
        char name[256];                 // 이름
        int hoursWorked;                // 근무시간
}

void main()
{
        int totalPay;
        struct _person * pPerson;
        pPerson = (struct _person*) malloc(sizeof(struct _person));
        if (pPerson) {
            pPerson->hoursWorked = 100;
            strcpy(pPerson->name, “Steve”);
            totalPay = pPerson->hoursWorked * 20000;
            printf(“Total payment for %s is %d”, pPerson->name, totalPay);
         }
        free(pPerson);
}
```

# C++ 프로그램

```
class Person {
private:
        std::string name;                    // 이름
        int hoursWorked;                     // 근무시간
public:
        Person(std::string n) : name(n), hoursWorked(0);
        std::string getName() { return name; }
        void setHoursWorked(int h) {hoursWorked = h; }
        int calculatePay() { return 20000*hoursWorked; }
};
void main(){
    Person * pPerson = new Person(“Steve”);
    if (pPerson) {
        pPerson->setHoursWorked(100);
        int totalPay = pPerson->calculatePay();
        std::cout << “Total payment for “ << pPerson->getName() << “ ”
                totalPay << std::endl;
    }
    delete pPerson;
}
```

# C# 프로그램

```
using System;
class Person {
        public string name;             // 이름
        private int hoursWorked;        // 근무시간
        public Person() : this (“”, 0) {}
        public Person(string name, int hoursWorked) {
                this.name = name;
                this.hoursWorked = hoursWorked;
        }
        public int CalculatePay() {     return 20000*hoursWorked; }
}
class PersonApp {
    public static void Main(strings [] args) {
        Person person = new Person(“Steve”, 200);
        Console.WriteLine(“Total payment for {0} is {1}”,
                                person.name, person.CalculatePay());
    }
}
```

# OOP (Object-Oriented Programming)

## 객체 지향 프로그래밍

- 프로그래밍하는 스타일
- 기존의 절차 중심 프로그래밍(procedural programming 또는 structured programming)의 재사용성을 개선한 방법
- 컴퓨터 프로그램을 단순히 데이터와 처리 방법으로 나누는 것이 아니라, 프로그램을 수많은 '객체(Object)'라는 기본 단위로 나누고
  이 객체들이 서로 메시지를 주고 받으며 데이터를 처리할 수

**객체**

하나의 역할을 수행하는 '메소드와 변수(데이터)'의 묶음

# OOP (Object-Oriented Programming)

## 소프트웨어의 생산성 향상

- 객체 지향 언어는 상속, 다형성, 객체, 캡슐화 등 소프트웨어 재사용을 위한 여러 장치가 내장되어 있음
- 소프트웨어의 재사용과 부분 수정을 통해 소프트웨어를 다시 만드는 부담을 대폭 줄임으로써 소프트웨어의 생산성이 향상됨

OOP

# OOP (Object-Oriented Programming)

## 실세계에 대한 쉬운 모델링

- 과거
  - ✓수학 계산/통계 처리를 하는 등의 처리 과정, 계산 절차가 중요
- 현재
  - ✓컴퓨터 프로그래밍이 산업 전반에 활용됨으로써 실세계에서 발생하는 일을 프로그래밍
  - ✓실세계에서는 절차나 과정보다 일과 관련된 물체(객체)들의 상호 작용으로 묘사하는 것이 용이
- 실세계의 일을 보다 쉽게 프로그래밍하기 위한 객체 중심의 객체 지향 언어 탄생

# PP vs OOP

## 절차 중심 프로그래밍 (Procedural Programming)

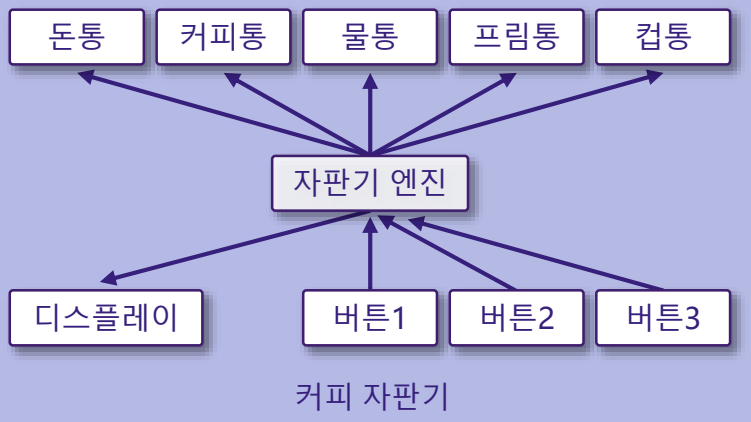
- 문제를 해결할 때 순서를 정해서 단계별로 작업하듯이, 코드를 주어진 절차 순서대로 실행
- 복잡해지지 않고 재사용성을 높이기 위해 함수(function) 또는 프로시듀어(procedure)라는 작은 단위로 나누어서 작성
- 절차와 데이터가 분리되어 있는 경우가 많고, 관리 어려움

## 객체 지향 프로그래밍 (Object Oriented Programming)

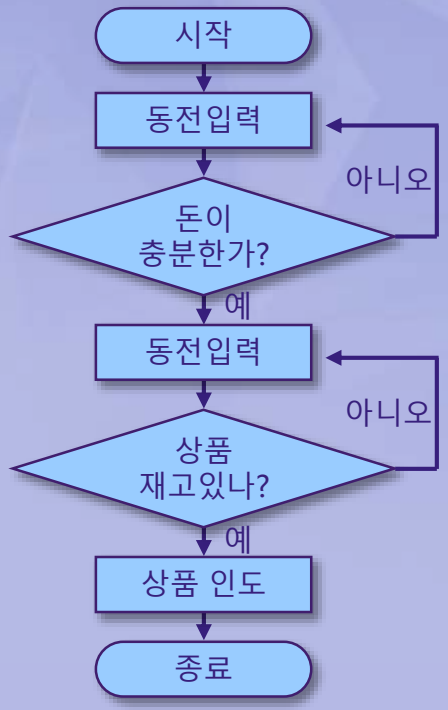
- 객체 지향 프로그래밍은 데이터와 코드를 객체로 함께 구성해서
  한 개의 자료형으로 취급함
- 프로그램을 실제 세상에 가깝게 모델링
- 컴퓨터가 수행하는 작업을 객체들 간의 상호 작용으로 표현
- 클래스 혹은 객체들의 집합으로 프로그램 작성

# PP vs OOP

## 커피 자판기 예제




객체지향적 프로그래밍의
객체들의 상호 관련성

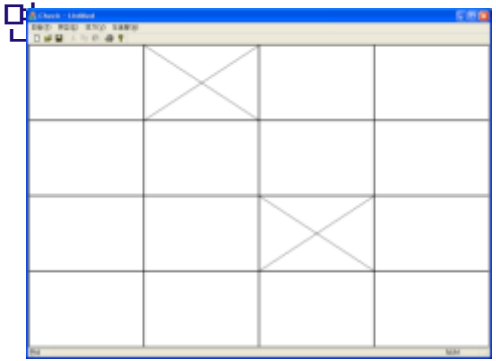



절차중심적 프로그래밍의
실행 절차

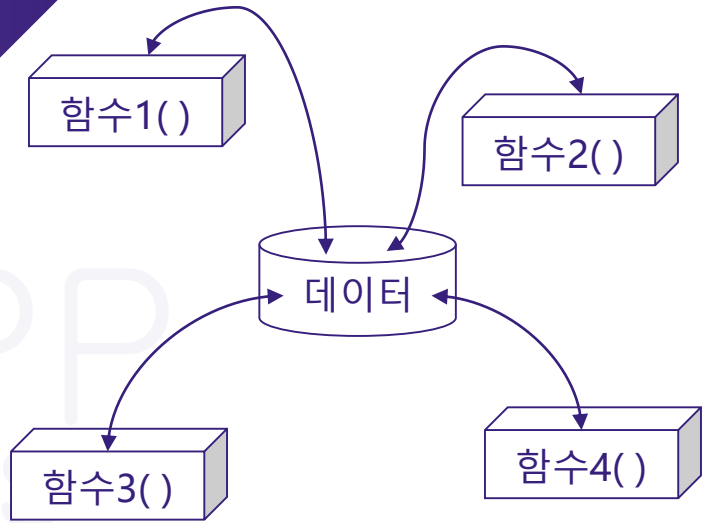
# PP vs OOP

## 클릭한 곳에 X표하는 프로그램 예제

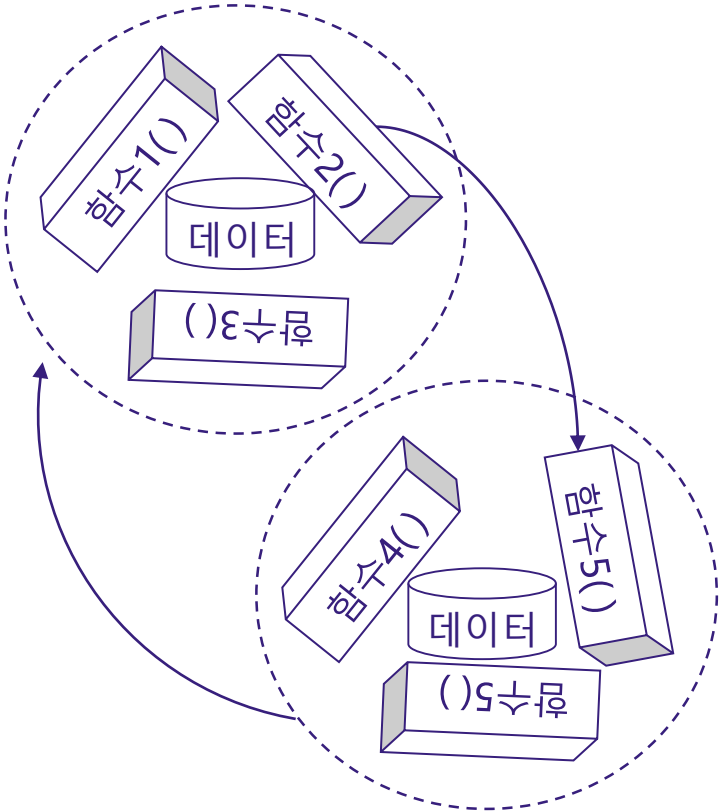
- 구조적 프로그래밍
  - ✓ 마우스가 클릭되면, 클릭된 점의 좌표를 계산
  - ✓ 클릭된 점의 좌표가 몇 번째 격자인지 계산
  - ✓ 그 격자의 모서리 점을 계산하여 대각선을 그리기
- 객체지향 프로그래밍: 각각의 격자를 하나의 오브젝트로 처리
  - ✓ 마우스가 윈도우에 클릭되면 자기자신 윈도우 전체에 대각선을 그리기



# PP vs OOP




PP에서는 데이터와 알고리즘이
묶여 있지 않음




OOP는 데이터와 알고리즘이
묶여 있음

## 자료 추상화 (Data Abstraction)

- 캡슐화 (Encapsulation), 정보은닉 (Information Hiding)

## 캡슐화 (Encapsulation)

- 캡슐화의 필요성
  - ✓ 사용자는 오디오의 사용법만 파악
  - ✓ 사용자가 오디오의 반도체 동작원리나 내부회로까지 파악하여 내부부품을 떼었다 붙였다 하고 배선을 끊었다 이었다 하면 고장
- C 구조체
  - ✓ 변수만 캡슐화, 외부함수에 의해 수동적으로 제어
- C++/C# 클래스
  - ✓ 변수, 함수를 캡슐화, 내부함수를 통해 능동적으로 동작
  - ✓ public: 외부에서 보이는 변수
  - ✓ protected, private: 내부에만 보이는 변수

# C 언어의 구조

- 인터페이스 파일과 구현파일의 분리
  - 헤더 파일(.h): 인터페이스 파일
  - 소스 파일(.c): 구현 파일
- 헤더 파일
  - 함수 프로토타입만 보여줌
  - 블랙 박스 (정보의 은닉, 구현을 볼 수 없음)
  - 계약서 역할 (작업의 정의를 자세하고 정확하게 기술)

```
/* 헤더 파일의 예 */
typedef struct _Point {
        int _x;
        int _y;
} Point;

void setX(int x);
void setY(int y);
void move(int x, int y);
```

# C++ Class

- 클래스는 객체를 정의한 데이터 타입
  - 내부에서는 클래스의 멤버변수, 멤버 함수를 직접 접근 가능
  - 외부에서는 클래스의 인스턴스를 통하여 공개된 (public) 멤버 변수, 멤버 함수를 접근
- 클래스의 인스턴스(객체)를 생성하여 사용

```
//Point class 선언
class Point {
//데이터(멤버변수)
        int _x;
        int _y;
public:
//메소드(멤버함수)
        void setX(int x){ _x = x; }
        void setY(int y){ _y = y; }
        void move(int x, int y){…}
};
```

```
//Point 객체 사용
void main() {
//Point 클래스의 인스턴스 생성
    Point p;
//Point 객체의 함수 접근(호출)
    p.setX(100);
    p.setY(40);
    p.move(20, 50);
}
```

# C# Class

- 클래스는 기능과 속성의 집합으로 기존의 데이터 타입을 이용하여 새로운 데이터 타입을 만들어내는 것

- 클래스 (Class)
  - 객체를 정의한 데이터형
  - 객체를 구성하는 속성과 행위를 프로그램적 요소인 변수와 함수로 표현
  - 클래스에 데이터(일반적으로 private)와 메소드 (일반적으로 public) 정의

- 객체 생성 (Instance)
  - 클래스로부터 생성된 고유한 객체 (메모리 할당)

# C# Class

## 클래스에 데이터와 메소드 정의

```
class Point
{
    private int x, y;
    public void SetX(int x)    { this.x = x; }
    public void SetY(int y)    { this.y = y; }
    public void Move(int x, int y) { ….. }
}
class PointTest
{
    static void Main(string [] args) {
        Point p = new Point();
        p.SetX(100);
        p.SetY(40);
        p.Move(20, 50);                    // X=120, Y=90
    }
}
```

# Object

- 이 세상 존재하는 모든 것(예: 사람, 자동차, 꽃, 등등) 은 객체 (Object)가 될 수 있음
- 객체의 구성요소
  - 독자성 (Identity)
  - 속성(Attributes, Properties, States, Data, Variables)
    - ✓객체를 구별시키는 상태 값을 나타내는 데이터
  - 행위 (Behaviors, Messages, Methods, Functions)
    - ✓ 객체 내부의 속성 값을 변경하거나 조작하는 기능
    - ✓ 외부의 다른 객체에게 영향을 주거나 받는 동적인 기능

**전자레인지**

상태: 직육면체
버튼, 문
마이크로웨이브
등

행동: 전원이 들어오다.
전원이 나가다.
조리하다.
타이머를 맞추다.
등

# Object-Oriented Programming 특징

- 캡슐화 (Encapsulation)
- 상속 (Inheritance)
- 다형성 (Polymorphism)

# Encapsulation

## 캡슐화(Encapsulation)

- 자세한 내부 사정을 드러내지 않고 외부에 보이는 부분만으로 충분히 사용할 수 있도록 만드는 작업

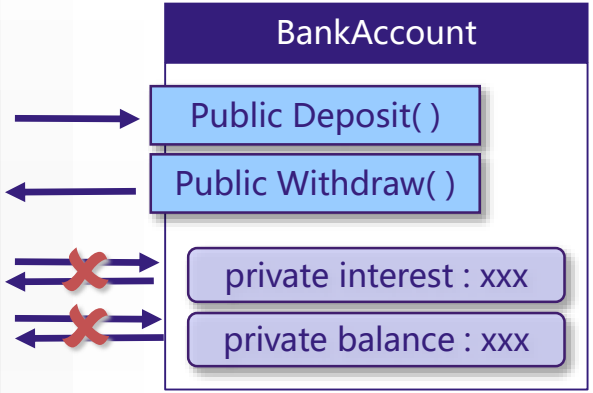
## 객체 지향 프로그래밍에서의 캡슐화

- 관련된 데이터와 알고리즘을 하나의 덩어리로 묶는 것
- 메소드(함수)와 데이터를 클래스 내에 선언하고 구현
- 외부에서는 공개된 메소드의 인터페이스만 접근 가능
  - ✓외부에서는 비공개 데이터에 직접 접근하거나 메소드의 구현 세부를 알 수 없음
- 객체 내 데이터에 대한 보안, 보호, 외부 접근 제한

# Encapsulation & Information Hiding

## 정보 은닉

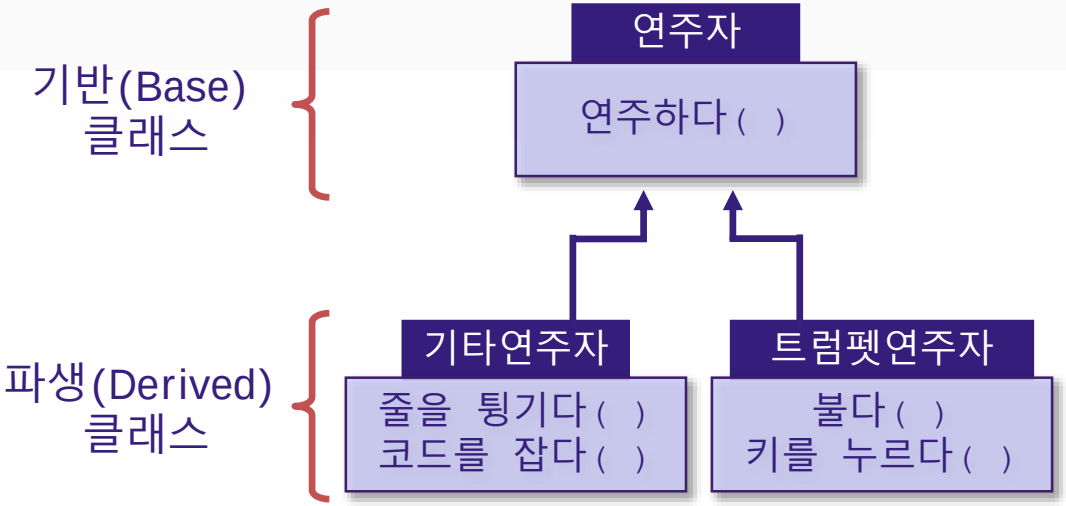
- 정보 은닉(**information hiding**)은 객체를 캡슐로 싸서 객체의 내부를 보호하는 하는 것이다. 즉, 객체의 실제 구현 내용을 외부에 감추는 것임
- 상태정보를 나타내는 변수 데이터는 **private**으로 지정하여 외부에 공개하지 않음
- 행동정보를 나타내는 메소드를 **public**으로 지정하여 외부에서 접근할 수 있게 함



# Inheritance

## 상속 (Inheritance)

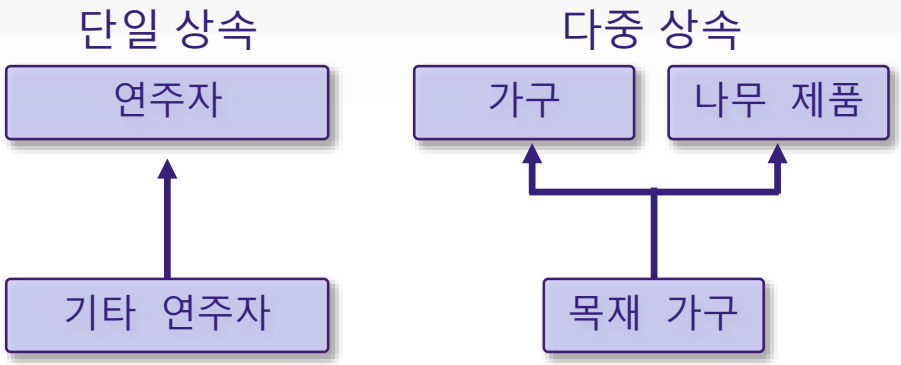
- 이미 작성된 클래스(부모 클래스)를 이어받아서 새로운 클래스(자식 클래스)를 생성하는 기법
- 기존의 코드를 재사용 또는 기존 타입의 확장



# Inheritance

## 단일상속과 다중상속

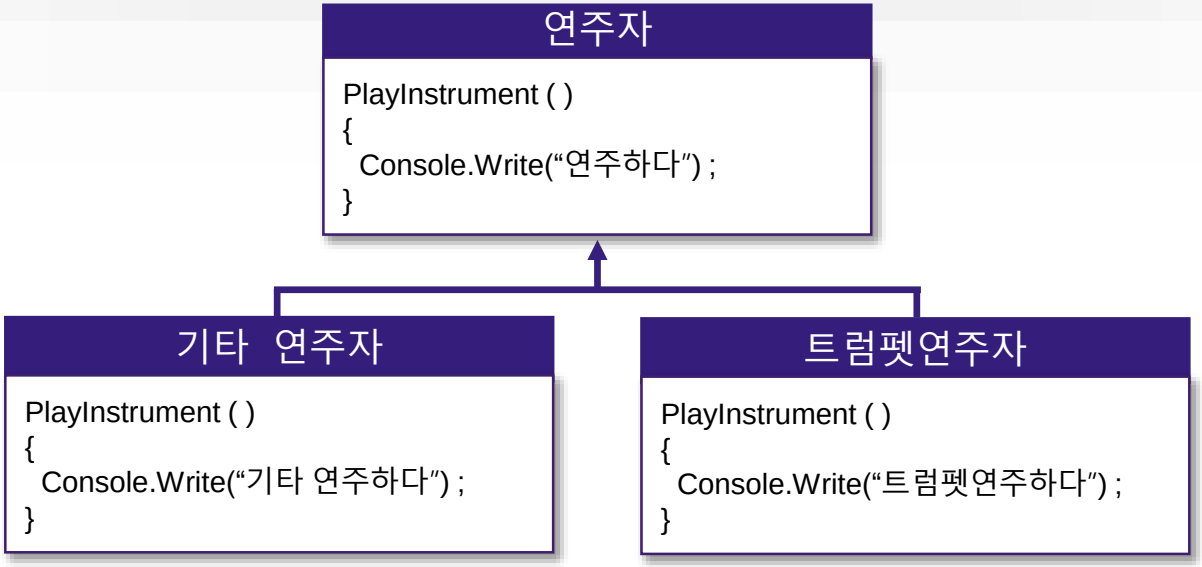
- 상위 기반 클래스가 하나인 경우는 단일상속
- 상위 기반 클래스가 2개 이상인 경우 다중상속
- **C# 언어에서는 다중상속을 지원하지 않음**



# Polymorphism

## 다형성 (Polymorphism)

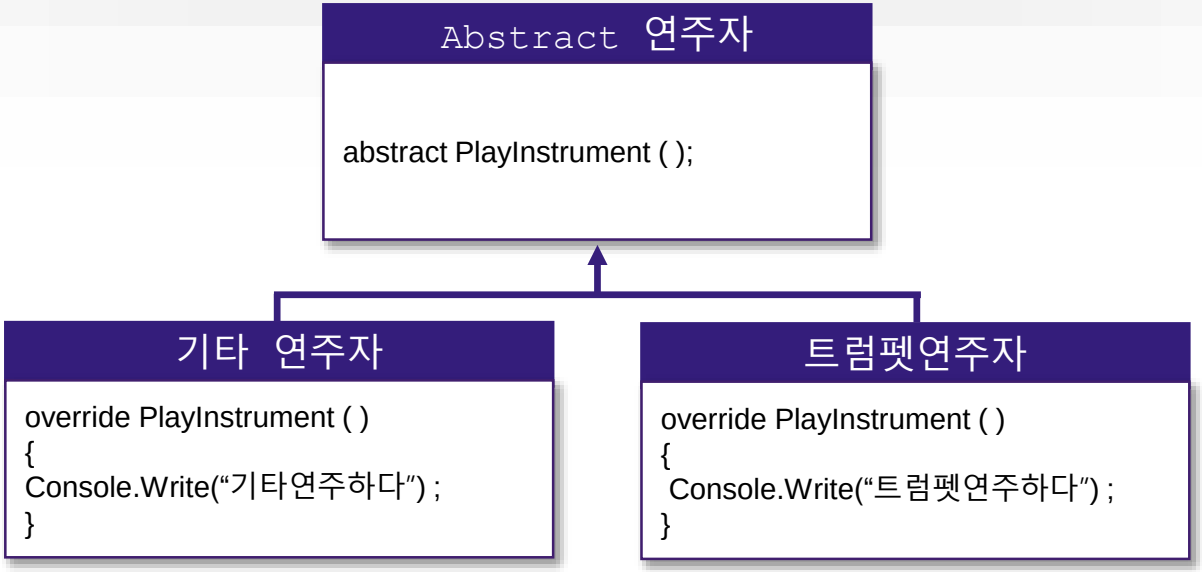
- 동일한 이름으로 많은 상황에 대처하는 기법
- 기반 클래스와는 다른 형태를 파생클래스에서 구현가능
  - 메소드 오버라이딩



# Abstract Class

## 추상클래스 (Abstract Class)

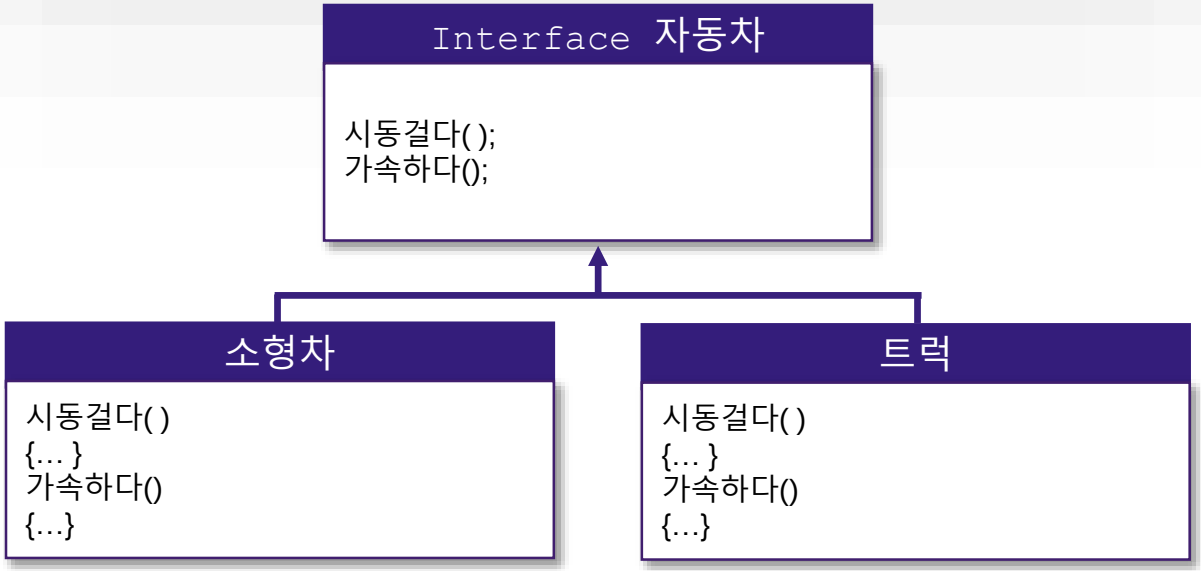
- 클래스 설계적인 측면에서 파생클래스에서 구현해야 할 내용들을 정의하기 위해 사용됨
- **C#에서 abstract 키워드를 사용**



# Interface

## 인터페이스 (Interface)

- 클래스에 대한 설계도
- 추상클래스와 기능적인 유사점
- **C#에서 interface 키워드를 사용**



# Class, Object, In

## 클래스 (Class)

- 객체(Object)의 속성과 행위 선언
- 객체의 설계도 혹은 틀

## 객체

- 클래스의 틀로 찍어낸 실체
  - ✓ 메모리 공간을 갖는 구체적인 실체
  - ✓ 클래스를 구체화한 객체를 **인스턴스(Instance)**라고 부름
  - ✓ 객체와 인스턴스는 같은 뜻으로 사용

# Class 정의

## 문법

```
class 클래스명
{
        //클래스 멤버 필드 & 메소드
}
```

# Class 정의

## 클래스 멤버

### 필드 (Field)

클래스 범위에서 선언된 변수. 객체의 상태정보 저장

### 메소드 (Method)

객체의 행동특성 정의

### 연산자 (Operator)

오버로드 된 연산자는 형식멤버로 간주

### 생성자 (Constructor)

클래스 인스턴스 생성 및 멤버 초기화

### 속성 (Property)

필드와 같이 상태정보를 저장하며 내부적으로 상태에 대한 접근 가능한 메소드 제공

- Smart Field

### 인덱서 (Indexer)

객체의 배열화한 형태

- Smart Array

### 이벤트 (Event)

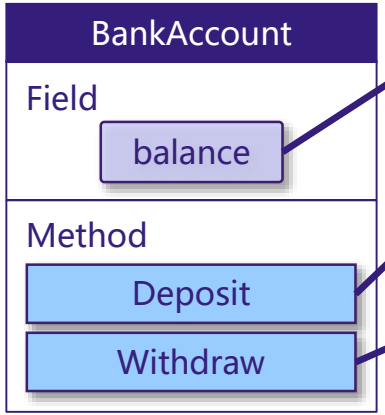
어떤 사건이 발생하였을 때 사용자에게 해당사실을 알려주는 방법

# Class 정의

```
class Customer {
    private string name;                    // 필드
    private int age;                        // 필드
    public event EventHandler NameChanged;      // 이벤트
    public Customer() { // 생성자
        name = string.Empty;
        age = -1;
    }
    public string Name { // 속성
        get { return this.name; }
        set {
            if (this.name != value) {
                this.name = value;
                if (NameChanged != null) {
                    NameChanged(this, EventArgs.Empty);
                }
            }
        }
    }
    public int Age { // 속성
         get; set;
    }
    public string GetCustomerData() { // 메소드
        return string.Format("Name: {0} Age: {1}", this.Name, this.Age);
    }
}
```

# Class 정의

## BankAccount 예제




```
class BankAccount
{
    private decimal balance;                    //잔액
    private string name;                        //이름
    static double interest = 0.7;               //이율
    public void Deposit(decimal amount)
    {
            balance = balance + amount ;
    }
    public void Withdraw(decimal amount)
    {
            balance = balance – amount ;
    }
    public static InterestRate (double in)  //이자계산
    { … }
}
```

Static fields
객체에서 공유할 데이터

Static methods
Static fields만 접근 가능한 행위

# Field

- C++의 데이터 멤버를 의미하고 C++와 차이가 없음
  - C# 클래스 멤버 필드는 정의되지 않았을 시 자동으로 **private**
- 필드는 명시적으로 초기화 하지 않을 경우,
  수치형은 0. 부울은 false, 참조형은 null로 초기화
- C#에서도 static 필드를 제공
  - C++에서는 일반필드를 참조하듯이 객체를 통하여 정적필드를
    참조 가능했으나 C#에서는 반드시 클래스명.정적필드 형태로 참조
- C#에서는 readonly라는 읽기전용 필드를 제공
- #에서는 const 키워드를 사용하면 필드 또는 지역 변수의 값을
  상수(constant)로 즉 수정할 수 없도록 지정
  - 클래스당 하나만 상수필드가 할당
  - 정적필드와 마찬가지로 클래스명.상수명 형태로 참조

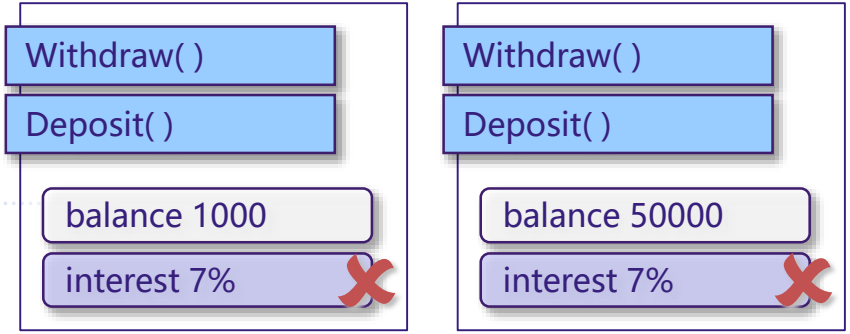
# Instance Field vs. Static(Class) Field

## Instance field

- 객체의 현재상태를 저장할 수 있는 자료
- 객체 생성시 메모리 공간 할당

## Static(Class) field

- **전역 데이터, 공유 데이터**로 클래스 로드 시 공간을 할당함
- 클래스당 하나만 정적 필드가 할당
- 객체의 개수를 카운트하거나 통계 값들을 저장하는 경우 사용함
- **C#에서는 클래스 이름으로 접근해야 함**
  - 클래스명.정적필드명



# Readonly Field

## Readonly field

- 객체 생성 시에 지정된 값을 변경할 수 없게 해 줌
- C#에서 **readonly**로 선언된 필드의 값은 선언 시에 초기값을 확정하거나 아니면 생성자 부분에서 값을 확정할 수 있음

```
class ReadonlyClass {
        public int x;                          // instance field
        public readonly int y = 1;     // readonly field initialization
        public readonly int z;                 // readonly field
        public ReadonlyClass() { z = 24; }
        public ReadOnlyClass(int x, int y, int z) { this.x = x; this.y = y; this.z = z; }
        static void Main() {
                ReadonlyClass r1 = new ReadonlyClass(1, 2, 3);
                ReadonlyClass r2 = new ReadonlyClass();
                r2.x = 4;
                //r2.y = 5;              // Error
                //r2.z = 6;              // Error
}   }
```

# Const Field

## Const field

- C#에서는 const 키워드가 C++와는 약간 다르게 사용
- C#에서 const 키워드는 필드 또는 지역변수의 값을 상수로 (즉, 수정할 수 없도록) 지정
- 상수는 일조의 정적 필드이기 때문에 클래스당 하나만 할당
- 상수를 참조할 때에는 다른 정적 필드와 마찬가지로, 클래스명.상수명 형태로 참조
- static 키워드는 const 키워드와 같이 사용할 수 없음

```
public const int x = 1, y = 2, z = 3;        // const field (must initialize)
public const int w = x + 100;                // const field (must initialize)
```

# Readonly vs Const Field

## Const field

- const 상수는 컴파일 타임(compile-time) 상수이다.
- const 상수는 선언하는 순간부터 static이 된다.
- const 상수를 선언함과 동시에 초기화를 해주어야 한다.
- const 상수는 컴파일시 값이 결정 되어져 있어야 한다.

## Readonly field

- readonly 상수는 런타임(run-time) 상수이다.
- readonly 상수는 static 키워드를 사용하면 static 상수가 된다. 사용하지 않으면 일반상수가 된다.
- readonly 상수는 const 키워드를 사용하는 것처럼 반드시 초기화 될 필요 없다.
- readonly 상수는 생성자를 통해서 런타임시 값이 결정될 수 있다. 한번 값이 결정되면 다시는 그 값을 변경할 수는 없다.

# Member Field vs Local Variable

- 지역변수는 메소드 안에 선언
- 메소드의 매개 변수도 지역 변수의 일종

```
public class Box {
        private int width; // 클래스 멤버필드는 기본값으로 초기화
        private int length; // 클래스 멤버필드는 기본값으로 초기화
        private int height; // 클래스 멤버필드는 기본값으로 초기화

        public int GetVolume() { // 메소드
          int volume = width * length * height; // volume은 지역변수는 초기화 후 사용가능
          return volume;
    }
}
```

# Local Variable

- 지역 변수는 멤버필드와는 달리 자동으로 초기화되지 않으며, 초기화되지 않은 상태로 사용하게 되면 컴파일 시 오류 메시지가

```
class MethodTest {
        static void Main(string[] args) {
                int x = 3;
                //int y; // 오류발생 => int y = 0;으로 초기화 필요
                int z;   // OK
                MethodTest t = new MethodTest();
                t.Func(ref x, ref y);
                t.Func2(ref x, out z);
        }
        void Func(ref int p, ref int q) {
                q = p * p;                p = 2 * q;
        }
        void Func2(ref int p, out int q) {
                q = 2 * p;
}   }
```

# Method

- 객체의 동작을 정의
- 형식

```
[접근 한정자] [static/abstract/virtual/override] 반환값유형 메소드명
([매개변수들]) {
        // 지역변수 선언 및 메소드 구현
        return(반환값);
}
```

| static | abstract | virtual | override |
|---|---|---|---|
| class method, static member field 조작을 위한 메소드 | 추상클래스에서 선언될 추상 메소드 | 상속관계에서 상위클래스에 정의되는 메소드, 재정의(override)가 가능함을 명시하는 메소드 | 하위클래스에서 메소드를 재정의(override)할 때 사용, 상위 클래스의 virtual 메소드와 매칭 |

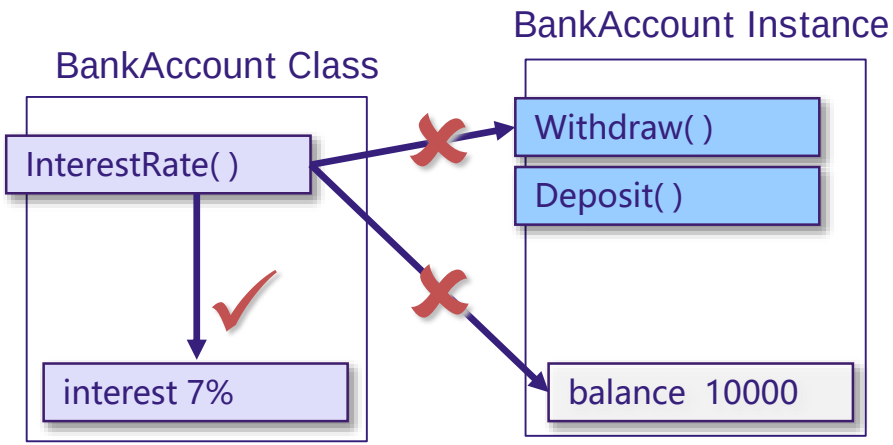
# Instance Method vs. Static Method

```
using System;
class StaticTest {
   public int x;        //instance field
   public static int y;//static field
   public void Test() {   // instance method
      x = 1;                // this.x = 1
      y = 1;                // StaticTest.y = 1
   }
   // static method
   public static void Test2() {
      StaticTest t = new StaticTest();
      t.x = 2;
      //x = 2;              // Error, Object reference
      y = 2;      // OK,
   }
   // static method
   public static string AddString(string str)  {
      str += " TEST!!";
      return str;
   }
}
```

```
class StaticTestApp {
public static void Main() {
   StaticTest t = new StaticTest();

   t.x = 10;        // OK
   // t.y = 10;    // Error, static member field
   StaticTest.y = 10;  // OK

   t.Test();        // OK x=1,y=1
   //t.Test2();        // Error, static method
   StaticTest.Test2();  // OK, but x=1, y=2

   Console.WriteLine("{0}, {1}", t.x, StaticTest.y);
   string str = “TEST";
   Console.WriteLine(StaticTest.AddString(str));
   // 출력값, TEST TEST!!
 }
}
```

# Static (Class) Method

- 정적(클래스) 데이터를 접근할 수 있는 메소드
- Instance field는 접근 불가능



# Method Parameter

- C#은 가변길이 매개변수 지원함
  - 메소드의 인수 목록 끝 부분에 배열로 선언하고 가변 길이임을 나타내기 위해 `params` 키워드를 추가로 붙여 줌
- C# 4.0부터 optional parameter 지원함
  - 메소드의 매개변수가 기본값을 갖고 있다면, 멤버함수 호출 시 이러한 매개변수 생략을 허용함
- C# 4.0부터 named parameter 지원함
  - 위치와 상관없이 매개변수를 지정하여 전달할 수 있게 함

# Method Parameter

```
static int[ ]  GetArray(params int[ ]  array) { return array; }
static int Sum(int a = 2, int b = 3) { return a + b; }
static void Main(string[ ]  args) {
    int[ ]  arr = GetArray(1, 2, 3, 4, 5); // params
    Console.WriteLine(Sum()); // 2 + 3 = 5 optional parameter
    Console.WriteLine(Sum(3)); // 3 + 3 = 6 optional parameter
    Console.WriteLine(Sum(b: 10, a: 5)); // 5 + 10 = 15 named parameter
}
```

# Method Parameter

- **in, ref** 또는 **out** 없이 메서드에 대해 선언된 매개 변수는 **값**으로 호출된 메서드에 전달됨
- **in** 매개변수 한정자는 C# 7.2 이상부터 사용가능
- in, ref, out 매개변수 한정자는 함수 시그니처의 일부로 간주되지만, 단일 형식으로 선언된 멤버는 시그니처에서 in, ref, out 만 다를 수 없음
  - `void Method1(int i, in int j)`와 `void Method1(int I, ref int j)`는 메소드 오버로딩 불가 컴파일 에러
  - `void Method1(int i, in int j)`와 `void Method1(int I, int j)`는 메소드 오버로딩 가능

```
static void DoSomthing(in int number) { // in 키워드사용 number는 read-only
    // number = 10; // compile error
}
static void Main(string[ ]  args) {
    DoSomthing(20);
}
```

# Method Parameter

- out 매개변수의 경우 미리 선언해야 했는데 C# 7.0 이상부터 간결하게 사용가능

```
static void GetData(out int x, out int y) {
    x = 10;
    y = 20;
}
static void Main(string[ ]  args) {
    // C# 7.0이전에는 미리 선언해야 했음
    int a, b;
    GetData(out a, out b);
    Console.WriteLine(a + “, “ + b);

    // C# 7.0이후
    GetData(out int a, out int b); // a, b 변수는 메소드 호출 후 계속 사용
    Console.WriteLine(a + “, “ + b);
}
```

# Parameter Passing

## Pass by value

- 값 형식(value type)은 copy of value 를 전달하는 방식
- 참조 형식(reference type)은 copy of reference를 전달하는 방식
- **Method(int x, int y), Method (int[] a)**

## Pass by reference

- 참조에 의한 전달방식을 사용
- **ref** 키워드 사용
  (메소드 정의 / 메소드 호출 )
- **호출 전에 매개변수 초기화**
- 메모리가 할당된 참조 형 데이터를 매개변수로 사용
- **Method(ref int x, ref int y), Method (ref int[] a)**

## Pass by output

- 출력에 의한 전달방식을 사용
- 값을 반환 받을 때만 사용
- **out** 키워드 사용
- **Method(out int x, out int y), Method(out int[] a)**

# Parameter Passing

```
//Pass by value (value type)
using System;
class ParamPass
{
   static public void Increase(int i)
   {
      i++;
      Console.WriteLine("내부 i={0}", i);
   }
   static void Main(string[] args)
   {
      int i = 10;
      Console.WriteLine("호출전 i={0}", i); //10
      // Pass by value (value type)
      Increase(i); // 11
      Console.WriteLine("호출후 i={0}", i); //10
   }
}
```

```
//Pass by value (reference type)
using System;
public class Ref {
        public int i = 10;
}
class ParamPass {
   static public void Increase(ref i)
   {
      i.i++;
      Console.WriteLine("내부 i={0}", i.i);
   }
   static void Main(string[] args)
   {
      Ref r = new Ref();

      Console.WriteLine("호출전 r.i={0}", r.i); //10
      // Pass by value (reference type)
      Increase(r); //11
      Console.WriteLine("호출후 r.i={0}", r.i); //11
    }
}
```

# Parameter Passing

```
using System;
class ParameterPassing {
    static int Sum(int a, int b) {
      int sum = 0;
      sum = a + b;
      return (sum);
   } // pass-by-value
   static void Swap(int a, int b) {
      int t = a;
      a = b;
      b = t;
   } // pass-by-value
   static void RefSwap(ref int a, ref int b) {
      int t = a;
      a = b;
      b = t;
   } // pass-by-reference
   static void Divide(int a, int b, out int result,
out int remainder) {
      result = a / b;
      remainder = a % b;
   } // pass-by-output
```

```
static void Main() {
   int x = 1;
   int y = 2;
   int out, remainder;
   Console.WriteLine(" x = {0}, y = {1}", x, y);
   Console.WriteLine("call Sum: x = {0}, y = {1},
                  sum ={2}", x, y, Sum(x,y));
   Swap(x, y);
   Console.WriteLine("call Swap: x = {0}, y = {1}", x, y);

   RefSwap(ref x, ref y);
   Console.WriteLine("call refSwap: x = {0}, y = {1}", x,
y);

   Divide(x, y, out result, out remainder);
   Console.WriteLine("call Divide: x = {0}, y = {1},
        result={2}, remainder={3}", x, y, result, remainder);
   }
}
```

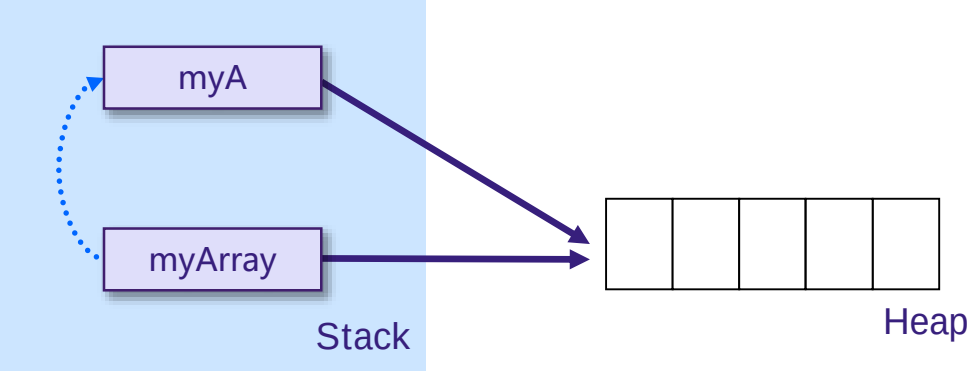
# Parameter Passing

```
//Pass by output (reference type)
using System;
class ArrayPass
{
   static public void FillArray(out int[ ]  myA)
   {
      myA = new int[5] {1, 2, 3, 4, 5};
   }
   static public void Main()
   {
      int[ ]  myArray;  // declaration (no initialization yet) OK

      // Pass by output
      FillArray(out myArray); //1,2,3,4,5

      // Display the array elements
      Console.WriteLine("Array elements are:");
      for (int i=0; i < myArray.Length; i++)
         Console.WriteLine(myArray[i]);
   }
}
```

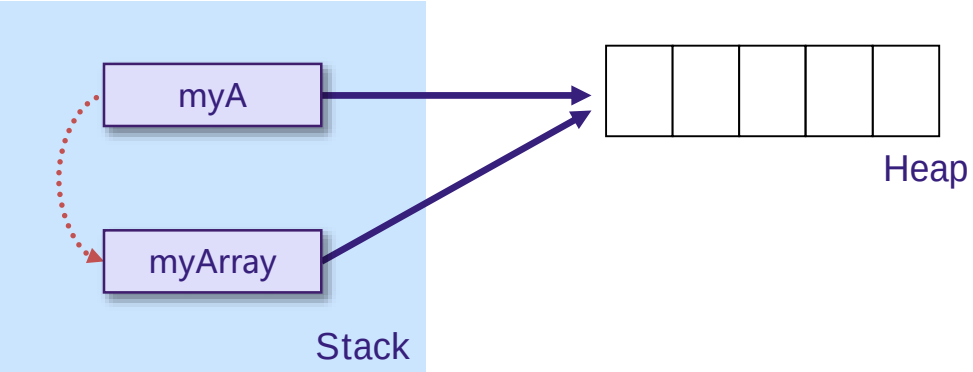
```
//배열 객체를 사용한 Pass by value (reference type)
using System;
class ArrayPass
{
   static public void FillArray(int[] myA)
   {
         for (int i=0; i < myA.Length; i++)
        myA[i] = i + 1;//fill the array elements
   }
   static public void Main()
   {
      int[] myArray = new int[5];
      // Pass by value
      FillArray(myArray); //1,2,3,4,5

      // Display the array elements
      Console.WriteLine("Array elements are:");
      for (int i=0; i < myArray.Length; i++)
         Console.WriteLine(myArray[i]);
   }
}
```

# Pass By Value & Pass By Output

FillArray(myArray);




FillArray(out myArray);



# Method Overloading

## 메소드 오버로딩 (Method Overloading)

- 메소드 이름은 같지만 매개변수가 다르게 정의
- 호출 시 메소드 매개변수 개수나 데이터 타입에 따라 구별되어 처리

```
class MethodOverload {
    int Max(int a, int b) {
        if (a > b) return a;
        else  return b;
    }
    double Max(double a, double b) {
        if (a > b) return a;
        else  return b;
    }
    static void Main(string[] args) {
        MethodOverload o = new MethodOverload();
        int x;      double y;
        x = o.Max(10, 50);                  // x=50
        y = o.Max(10.6, 50.3);  // y=50.3
    }
}
```

# Method Return

## 메소드는 호출자에 값을 반환할 수 있음

- 반환값 없이 메소드 실행 중단

```
return;
```

- 값(value type 또는 reference type)의 반환

```
return 값;     // 값으로 반환
```

  - ✓값 형식(value type)은 copy of value를 전달하는 방식
  - ✓참조 형식(reference type)은 copy of reference를 전달하는 방식

```
return ref 값; // 참조로 반환
```

- **C# 7.0부터 참조 반환**
  - ✓**ref** 키워드가 **함수 시그니처**에 사용되고 각 **return 키워드 뒤에 오면** 값이 호출자에 참조로 반환

# Method Return

```
class ReturnTest {
    int value = 100;
    public void SetValue(int v) {
        value = v;
    }
    public int GetValue() {
        return value;
    }
    public ReturnTest Copy1() {
        ReturnTest t = new ReturnTest();
        return t;
    }
    public ReturnTest Copy2() {
        return this;
    }
    public void Print(string name) {
      Console.WriteLine ("name={0},
                          value={1}",
                          name, value);
    }
}
```

```
class MethodReturnApp {
  public static void Main(){
   ReturnTest t1 = new ReturnTest();
   Console.WriteLine (t1.GetValue());
   ReturnTest t2 = t1.Copy1();
   Console.WriteLine (“Copy1 t1(new) to t2");
   t1.Print("t1");        t2.Print("t2");
   t1.SetValue(500); // t1.value = 500
   Console.WriteLine ("t1.SetValue(50)");
   t1.Print("t1");        t2.Print("t2");
   t2.SetValue(200); // t2.value = 200
   Console.WriteLine ("t2.SetValue(200)");
   t1.Print("t1");        t2.Print("t2");
   Console.WriteLine (“Copy2 t1(this) to t3");
   ReturnTest t3 = t1.Copy2(); //this copy
   t1.Print("t1");        t3.Print("t3");
   t3.SetValue(1000);
   Console.WriteLine ("t3.SetValue(1000)");
   t1.Print("t1");        t3.Print("t3");
   Console.ReadLine ();
  }
}
```

# Method Return

## 참조 반환 제한사항

- 참조 반환 값의 수명은 메소드 실행 이후까지 연장되어야 함
- 참조 반환 값은 리터럴 null 일 수 없음
- 참조 반환 값은 상수, 열거형 멤버, 속성의 값형식 반환값 또는 class/struct의 메서드일 수 없음

```
public static ref int FindNumber(int[] numbers, int target) {
    for (int i = 0; i < numbers.Length; i++) {
        if (numbers[i] >= target)
            return ref numbers[i]
    }
    return ref numbers[0];
}
static void Main(string[] args) {
    int[] numbers = new[] { 1, 3, 7, 15, 31, 63, 127, 255 };
    Console.WriteLine(String.Join(" ", numbers)); // 1 3 7 15 31 63 127 255
    ref int value = ref FindNumber(numbers, 16); // value = 31
    value *= 2; // value = 62
    Console.WriteLine(String.Join(" ", numbers)); // 1 3 7 15 62 63 127 255
}
```

# Method Return

```
public static ref string GetData(string[] arr, int index) {
    ref string str = ref arr[index]; // arr에 ref를 적용하여 동일한 레퍼런스 획득
    return ref str;
}
static void Main(string[] args) {
    var colors = new[] { "blue", "green", "yellow", "orange", "pink" };
    string color = colors[3];
    color = "Magenta";
    Console.WriteLine(String.Join(" ", colors)); // blue green yellow orange pink

    ref string color1 = ref colors[3];
    color1 = "Magenta";
    Console.WriteLine(String.Join(" ", colors)); // blue green yellow Magenta pink

    ref string color2 = ref GetData(colors, 3);
    color2 = "Magenta";
    Console.WriteLine(String.Join(" ", colors)); // blue green yellow Magenta pink
}
```

# Class
# 접근 제어

## 접근 지정자 (Access Modifier)

| 접근 지정자 | 내용 |
|---|---|
| public | • 외부에 모두 공개할 경우에 사용하는 접근 지정자<br>• 어느 서브 클래스나 인스턴스에서도 접근 가능 |
| private | • 같은 클래스 내에서만 접근이 가능<br>• 다른 클래스에서는 접근하지 못함(default) |
| protected | • 같은 클래스와 상속관계에 있는 파생 클래스에서만 접근할 수 있음<br>• 외부에서의 접근은 private |
| internal | • 동일한 물리적 파일 안에 있는 클래스에서만 접근 |
| protected internal | • protected와 internal의 조합<br>• 같은 물리적 파일 안의 파생 클래스 안에서만 접근이 가능 |
| private protected (C# 7.2) | • 같은 class나 해당 코드가 파생된 코드에서만 접근이 가능 |

# Class
# 접근 제어

## 접근 지정자 (Access Modifier)

| | Within class | Derived class (same assembly) | Non-derived class (same assembly) | Derived class (different assembly) | Non-derived class (different assembly) |
|---|---|---|---|---|---|
| public | Y | Y | Y | Y | Y |
| private | Y | N | N | N | N |
| protected | Y | Y | N | Y | N |
| internal | Y | Y | Y | N | N |
| protected internal | Y | Y | Y | Y | N |
| private protected (C# 7.2) | Y | Y | N | N | N |

# Class
# 접근 제어

## protected 키워드

- 파생클래스에서의 접근만 허용하고, 외부 클래스에서의 접근은 private처럼 제한

```
public class Car {
        protected int wheel = 4;
        protected void Move() {
            Console.WriteLine("바퀴 {0} 자동차가 굴러다닌다", wheel);
        }
}
public class CarApp {
    static void Main(string[] args) {
            Car myCar = new Car();
            //myCar.Move();            // 보호 수준 때문에 Car.Move()에 액세스할 수 없음
    }
}
```

# Class
# 접근 제어

## protected 키워드

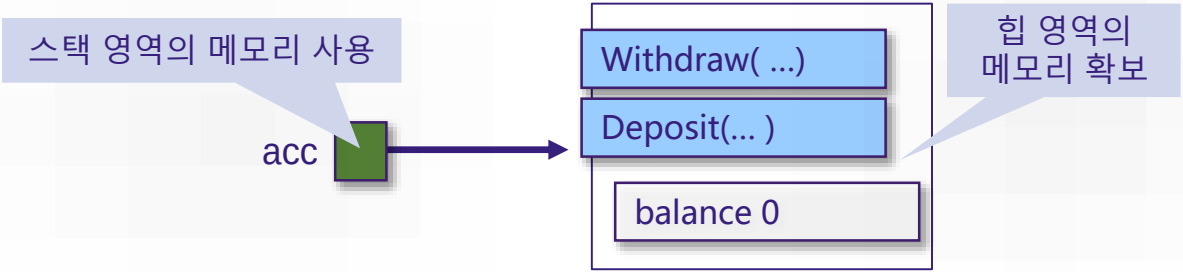
```
public class Sedan : Car {
        public void SedanMove() {
            Console.WriteLine("바퀴 {0} 승용차가 굴러다닌다", wheel);
        }
        static void Main(string[] args) {
            Sedan myCar = new Sedan();
            myCar.Move();              // Car 를 상속받았기 때문에 Move()
                                       // 메소드를 사용할 수 있음
            myCar.SedanMove();         // 자신의 메소드 사용가능
        }
}
```

# 객체 생성 (Instantiation)

## 객체의 생성

- 메모리에 객체를 만들어서 적재하는 것
- new와 생성자 (constructor)가 담당

```
class BankAccountApp {
    static void Main( )    {
        BankAccount acc = new BankAccount( );
        acc.Deposit(100000);
    }
}
```



# Constructor

## 생성자 (Constructor)

- 객체가 생성될 때 초기화를 위해 실행되는 메소드

## 생성자의 특징

- 생성자는 메소드
- 생성자 이름은 클래스 이름과 동일
- 생성자는 new를 통해 객체를 생성할 때만 호출됨
- 생성자도 오버로딩하여 여러 개 작성 가능
- 생성자는 리턴 타입을 지정할 수 없음
- 생성자는 하나 이상 선언되어야 함
  - ✓ 개발자가 생성자를 작성하지 않았으면 **컴파일러에 의해 자동으로 기본 생성자가 선언됨**
  - ✓ 기본 생성자를 디폴트 생성자(default constructor)라고도 함

# Default Constructor

## 기본 생성자(default constructor)

- 클래스에 생성자가 하나도 선언되지 않은 경우, 컴파일러에 의해 자동으로 생성
  - ✓매개 변수 없는 생성자
  - ✓아무 작업 없이 단순 리턴
- 디폴트 생성자라고도 부름

```
public class DefaultConstructor {
        int x;
        public void SetX(int x) { this.x = x; }
        public int GetX() { return x; }

        public static void Main(string [] args) {
                DefaultConstructor p = new DefaultConstructor();
                p.SetX(3);
        }
}
```

# Default Constructor

```
class DefaultConstructor {
        int x;
        public void SetX(int x) { this.x = x;}
        public int getX() { return x;}

        public DefaultConstructor() { }

        public static void Main(string [] args) {
                DefaultConstructor p = new DefaultConstructor();
                p.SetX(3);
        }
}
```

컴파일러에 의해
자동 삽입된 기본 생성자

컴파일러가 자동으로 기본 생성자를 삽입한 코드

# 기본 생성자가 자동 생성되지 않는 경우

- 클래스에 생성자가 하나라도 존재하면 기본 생성자가 자동 삽입되지 않음

```
class DefaultConstructor {
        int x;
        public void SetX(int x) { this.x = x; }
        public int GetX() { return x; }

        public DefaultConstructor(int x) {
                this.x = x;
        }
        public static void Main(string [] args) {
                DefaultConstructor p1= new DefaultConstructor(3);
                int n = p1.GetX();
                DefaultConstructor p2= new DefaultConstructor();
                p2.SetX(5);
        }
}
```

컴파일러가 기본 생성자를 자동 생성하지 않음

public DefaultConstructor() { }

컴파일 오류.
해당하는 생성자가 없음 !!!

# Constructor Overloading

- 오버로딩(Overloading)이란 같은 이름을 가진 메소드를 여러 개 정의 할 수 있는 프로그래밍 기법
  - 클래스의 기능을 표현하는 메소드를 여러 가지 변형된 모습으로 정의
  - 같은 이름을 가진 메소드라도 받아들이는 인자의 개수나 인자형을 달리하여 여러 개의 메소드를 정의하여 다른 메소드로 분류

```
public Date(int yy) {
        this.yy = yy ;
}
public Date(int yy, int mm, int dd) {
        this.yy = yy ;
        this.mm = mm ;
        this.dd = dd ;
}
```

# Constructor Overloading

## 생성자 오버로딩 (Constructor Overloading)

- 생성자를 여러 가지 형태로 정의해서 사용

인자가 없는 형태의 Date 기본 생성자

인자가 있는 형태의 Date 생성자

```
public Date()
{
        this.yy = 2004 ;
        this.mm = 1 ;
        this.dd = 1;
}

public Date( int yy, int mm, int dd)
{
        this.yy = yy ;
        this.mm = mm ;
        this.dd = dd ;
}
…
```

# Constructor Ini

## 생성자 초기화 목록 (Initializer List)

- 생성자들 사이의 정의가 비슷한 경우 코드를 간략하게 만들기 위해서 사용

**초기화 목록**
**(Initializer List)**

인자가 없는 형태의
생성자 Date( ) 를 사용하면
곧바로 초기화 목록에서
가리키는 생성자를 호출하게 된다.

```
<예>
public Date() : this (2004, 1, 1)
{
}

public Date( int yy, int mm, int dd)
{
        this.yy = yy ;
        this.mm = mm ;
        this.dd = dd ;
}
…
```

호출

# Structure Con

## Struct (구조체)의 생성자

- 구조체는 컴파일러가 기본 생성자를 기본적으로 생성하므로 기본 생성자를 사용자가 직접 정의하지 않음
- protected 접근 지정자를 사용할 수 없음
- 생성자에서는 구조체의 모든 멤버 값을 초기화 시켜줘야 함

구조체에 대한 생성자

```
...
struct Point
{
        public int x , y ;
        public Point ( int x, int y)
        {
                this.x = x ;
                this.y = y ;
        }
}
...
```

# this 키워드

## this 키워드

- 클래스 내에서 클래스가 갖고 있는 멤버변수, 메소드를 직접 참조할 수 있는 변수 (자기 참조 변수)

```
// 인자가 두 개인 생성자
public Car(int maxspeed, string name)
{
    velocity = maxspeed;
    carName = name;
}
```

```
// this 키워드를 이용 변환
public Car(int velocity, string carName)
{
    this.velocity = velocity;
    this.carName = carName;
}
```

- 인자의 이름을 다르게 지정할 필요가 없음
- this 키워드는 메소드 내의 현재의 인스턴스를 가리킴

# Static Constructor

## Static 생성자

- 접근 지정자를 쓸 수 없음
- 일반적인 생성자와 같이 호출하지 않음
- 인자를 포함하지 않음

static 생성자

```
class Point
{
        private static int[] data ;

        static Point()
        {
                data = new int[1000];
                …

        }
}
…
```

# Static Constructor

## Private 생성자

- 정적 멤버만 포함하는 클래스에서 일반적으로 사용
- 클래스가 인스턴스화 될 수 없음을 분명히 하기 위해 private constructor를 사용

private 생성자

```
public class Counter {
    private Counter() { }
    public static int currentCount;
    public static int IncrementCount() { return ++currentCount; }
}
class TestCounter {
    static void Main() {
        // Counter aCounter = new Counter(); // Error
        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine("New Count: "+Counter.currentCount);
    }
}
```

# Static Constructor

## Protected 생성자

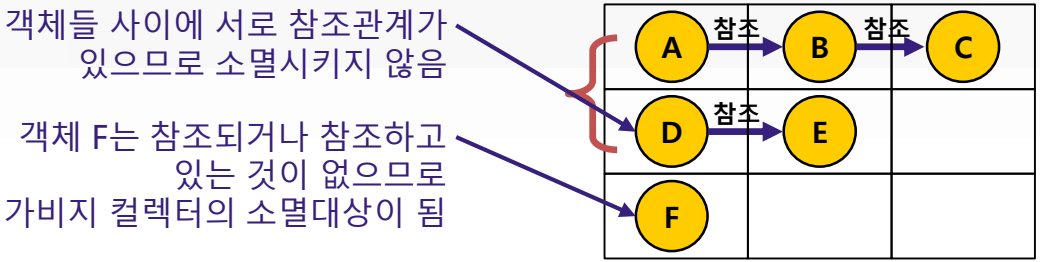
- Protected 생성자는 추상클래스(abstract class)에서 사용을 권함
- 추상클래스에는 protected 또는 internal 생성자를 정의함
- 추상클래스를 상속받은 파생클래스에서 추상클래스의 protected 생성자를 호출하여 초기화 작업을 수행함

```
public abstract class Shape {
    protected Shape(string name) { this.name = name; }
    private string name;
    public void Print() { Console.WriteLine(this.name); }
}
public class Triangle: Shape {    public Triangle(string name): base(name) {}
}
public class Rectangle: Shape {    public Rectangle(string name): base(name) {}
}
Shape s = new Triangle("삼각형");
s.Print(); // 삼각형
s = new Rectangle("직사각형");
s.Print(); // 직사각형
```

# Garbage Collection

## 가비지 컬렉션 (Garbage Collection)

- 가비지 컬렉터에 의해 자동 수행
- 객체가 할당된 후 더 이상 사용하지 않는 객체(즉, 참조가 없으면)에 대한 소멸작업을 함
- 객체의 필요성 여부를 참조의 유무로서 판단
- 힙 영역에 메모리가 모두 차서 더 이상 할당이 불가능한 경우 null을 참조하거나 정해진 범위를 참조할 경우 메모리 해제
- 안전하게 Dispose() 메소드를 이용하여 내부 리소스 정리



# Destructor

## 소멸자 (Destructor)

- 객체가 소멸될 때 필요한 정리 작업을 정의하는 부분
- "~클래스명"를 이용하여 소멸자 정의
- 컴파일러가 Finalize 메소드로 변환
- 접근 지정자를 사용하지 않음
- 반환값이 없음
- 인자를 포함하지 않음

소멸자

```
<예>
...
class SourceFile
{
    ~SourceFile()
    {
        ...
    }
}
...
```