객체 지향 개념
& 클래스와 객체

321190
2015년 가을학기
9/15/2015
박경신

Overview

- 객체지향의 기본적 원리 이해
- 클래스의 정의
- 객체의 생성, 소멸, 사용
- 클래스 생성자
- 가비지 컬랙션의 역할
- 클래스 필드, 메소드, 속성

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

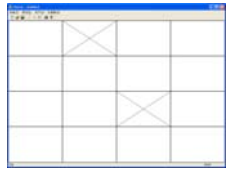
```
class Person {
private:
        std::string name;          // 이름
        int hoursWorked;           // 근무시간
public:
        Person(std::string n) : name(n), hoursWorked(0);
        std::string getName() { return name; }
        void setHoursWorked(int h) {hoursWorked = h; }
        int calculatePay() { 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());
    }
}
```

Object-Oriented Programming

- 객체지향 프로그래밍 (Object-Oriented Programming)
  - 프로그램 기본단위를 객체(object)로 해서 프로그램을 개발
  - 프로그램 기본단위: C는 함수이고, C++/C#은 클래스
- 구조적 프로그래밍 vs. 객체지향 프로그래밍
  - 예제: 클릭한 곳에 X표하는 프로그램



  - 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

Encapsulation

- 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 보여 줌
  - 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)

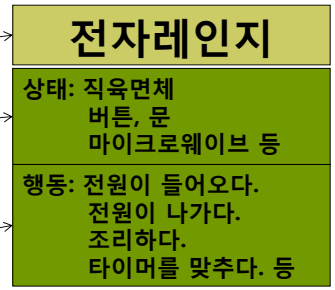
C# Class

- 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

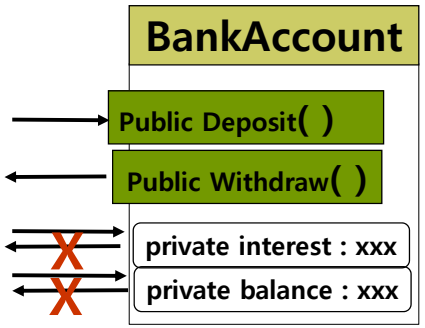


Object-Oriented Programming 기본원리

- 추상화 (Abstraction)
  - 중요한 것에만 초점을 맞추어 관리하는 개념
- 캡슐화 (Encapsulation)
  - 내부적인 것은 공개하지 않고 필요한 부분만 외부에 인터페이스를 제공
- 모듈화 (Modulation)
  - 기능단위를 작게 나누는 것
- 계층성 (Hierarchy)
  - 추상화 및 상속을 통해 객체 간의 관계를 계층화

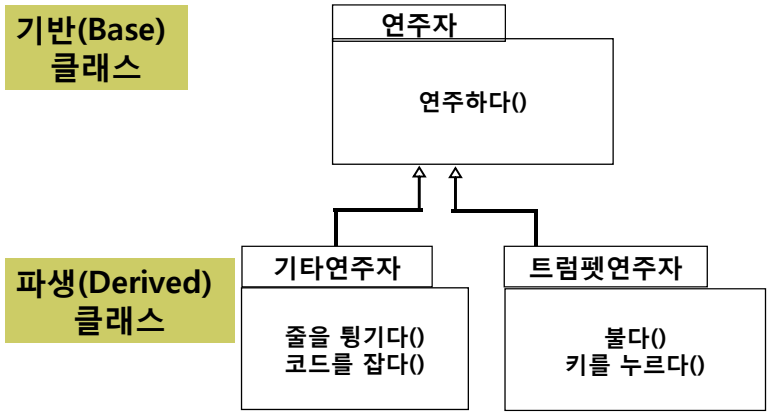
Encapsulation

- 캡슐화 (Encapsulation)
  - 상태정보를 나타내는 변수 데이터는 private으로 지정하여 외부에 공개하지 않음
  - 행동정보를 나타내는 메소드를 public으로 지정하여 외부에서 접근할 수 있게 함



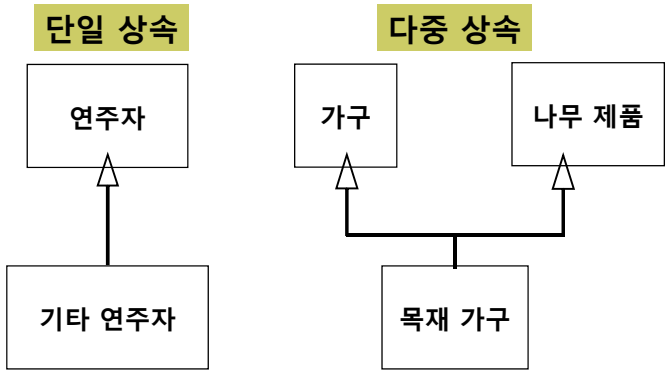
Inheritance

- 상속 (Inheritance)
  - 객체 간의 계층적 관계를 맺음으로써 코드의 재사용 및 간결성 추구



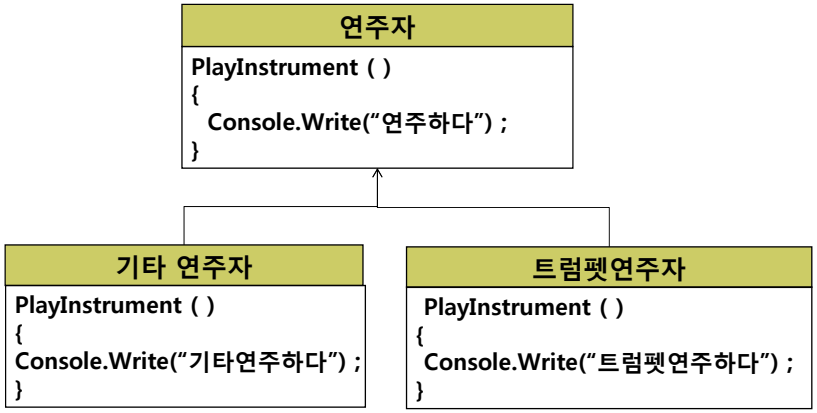
Inheritance

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



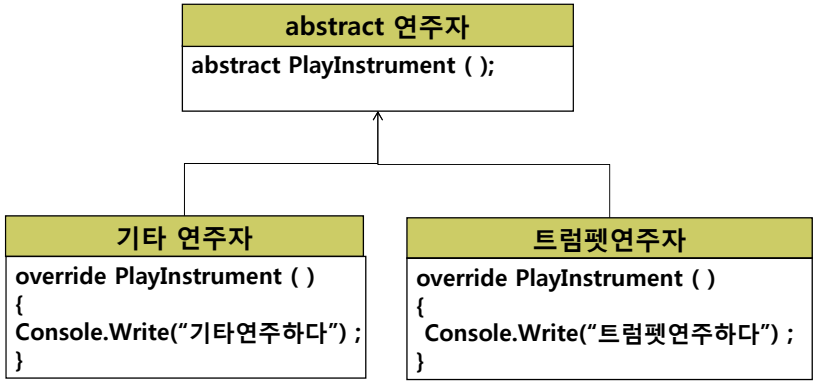
Polymorphism

- 다형성 (Polymorphism)
  - 기반 클래스와는 다른 형태를 파생클래스에서 구현가능



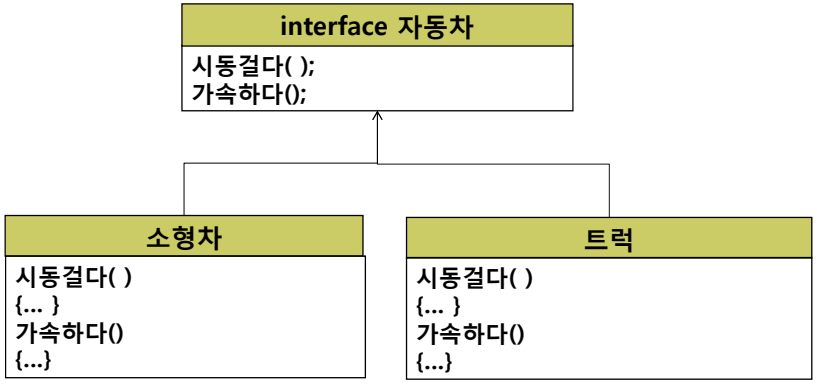
Abstract Class

- 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 정의

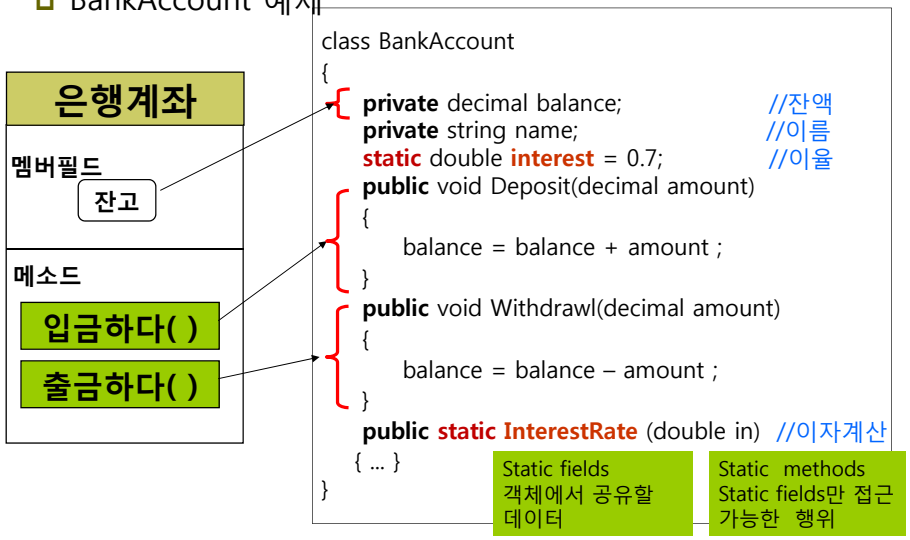
- 문법

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

- 클래스 멤버
  - **필드 (Field)** – 변수 혹은 멤버변수. 객체의 상태정보 저장
  - **메소드 (Method)** –객체의 행동특성 정의
  - **연산자 (Operator)** – 특정기능의 수행을 위한 기호
  - **상수 (Constant)** – 변하지 않는 값이나 숫자
  - **생성자 (Constructor)** – 클래스나 구조체의 인스턴스를 생성 혹은 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

- BankAccount 예제



Class 접근 제어

- 접근 지정자 (Access Modifier)

접근 지정자	내용
public	외부에 모두 공개할 경우에 사용하는 접근 지정자. 어느 서브 클래스나 인스턴스에서도 접근 가능.
private	같은 클래스 내에서만 접근이 가능. 다른 클래스에서는 접근하지 못함. (default)
protected	같은 클래스와 상속관계에 있는 파생 클래스에서만 접근할 수 있음. 외부에서의 접근은 private.
internal	동일한 물리적 파일 안에 있는 클래스에서만 접근
protected internal	protected와 internal의 조합. 같은 물리적 파일 안의 파생 클래스 안에서만 접근이 가능

Class 접근 제어

- protected 키워드
  - 파생클래스에서의 접근만 허용하고, 외부 클래스에서의 접근은 private처럼 제한

```
namespace CAR {
    public class Car {
        protected int wheel = 4;
        protected void Move() {
            Console.WriteLine("바퀴 {0} 자동차가 굴러다닌다", wheel);
        }
    }
}
```

보호 수준 때문에 'CAR.Car.Move()'에 액세스할 수 없음

Class 접근 제어

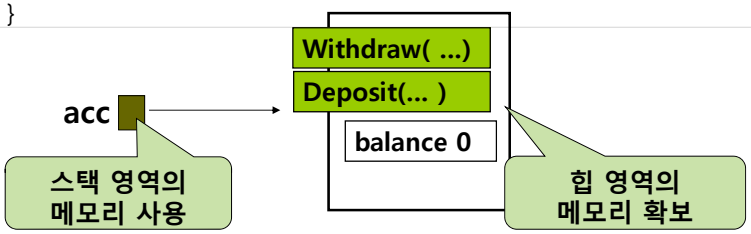
- protected 키워드

```
public class Sedan : Car {
        public void SedanMove() {
            Console.WriteLine("바퀴 {0} 승용차가 굴러다닌다", wheel);
        }
        static void Main() {
            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)
  - 클래스를 사용하기 위해 메모리에 객체를 생성하는 메소드
  - 만약 생성자 없이 메소드에서 클래스 객체를 생성할 경우 에러 발생
  - 생성자 선언

```
public class BankAccount()                    // Default Constructor
{
        this.name = name;
}
```

  - 클래스 객체 생성

```
// BankAccount 객체를 생성하여 메모리에 할당
BankAccount acc = new BankAccount();
```

Constructor

- 생성자 오버로딩 (Constructor Overloading)
  - 생성자를 여러 가지 형태로 정의해서 사용

인자가 없는 형태의 Date 기본 생성자

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

인자가 있는 형태의 Date 생성자

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

Method 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
  - 컴파일러는 호출된 메소드를 인자의 형태에 의해 판별

```
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 Initializer

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

Structure Constructor

- 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];
        ...
    }
}
...
```

Private 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);
    }
}
```

Protected 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.

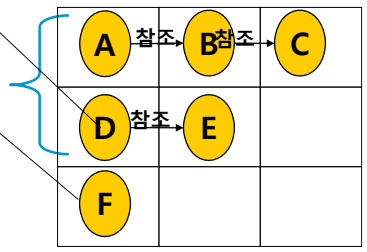
```
public abstract class Shape {
    protected Shape(string name) { this.name = name; }
    private string name;
    public void Print() { Console.WriteLine(this.name); }
}
public class Triangle: Shape {    public Triangle(string name): base(name) {}
}
public class Rectangle: Shape {    public Rectangle(string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

객체들 사이에 서로 참조관계가 있으므로 소멸시키지 않음

객체 F는 참조되거나 참조하고 있는 것이 없으므로 가비지 컬렉터의 소멸대상이 된다.



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

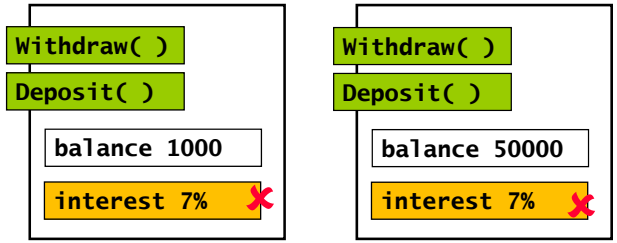
```
<예>
...
class SourceFile
{
    ~SourceFile()
    {
        ...
    }
}
...
```

Field

- C++의 데이터 멤버를 의미하고 C++와 차이가 없음
  - 다만, C#에서의 기본 재사용 단위가 어셈블리기 때문에 접근 한정자에 **internal**이 추가
- 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 **readonly**라는 읽기전용 필드를 제공
- C#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 데이터로 클래스 로드 시 공간을 할당함
  - 클래스당 하나만 정적 필드가 할당
  - 객체의 개수를 카운트하거나 통계 값들을 저장하는 경우 사용함
  - C#에서는 클래스 이름으로 접근해야 함 – 예) 클래스명.정적필드명



Readonly Field

- Readonly field
  - 객체 생성 시에 지정된 값을 변경할 수 없게 해줌
  - 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 필요없다.
  - readonly 상수는 생성자를 통해서 런타임시 값이 결정될 수 있다. 한번 값이 결정되면 다시는 그 값을 변경할 수는 없다.

Method

- 객체의 동작을 정의
- 형식

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

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

Method

- C#에서 메소드 내의 지역 변수
  - 필드와는 달리 자동으로 초기화되지 않으며, 초기화되지 않은 상태로 사용하게 되면 컴파일 시 오류 메시지가 발생

```
class MethodTest {
    static void Main() {
        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

- C#에서는 멤버함수 호출 시 해당 인수가 주어지지 않을 때 디폴트로 전달되는 기본인자(default parameter)도 제공하지 않음
- C#에서는 가변 길이 인수 (parameter)도 선언할 수 있음
  - 메소드의 인수 목록 끝 부분에 배열로 선언하고 가변 길이임을 나타내기 위해 **params** 키워드를 추가로 붙여줌

```
static long AddList(params long[] v) {
    long total. i;
    for (i = 0, total = 0; i < v.Length; i++)
        total  += v[i];
    return total;
}
static void Main() {
    long x = AddList(63, 21, 84);
}
```

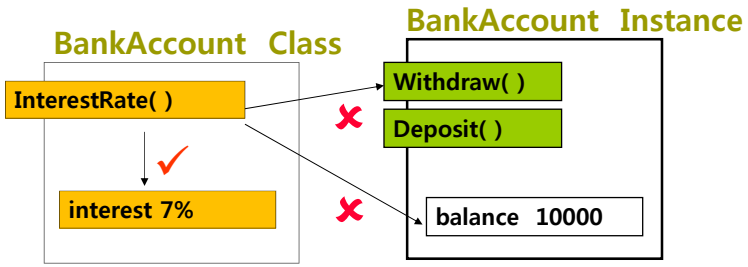
Instance Method vs. Static Method

```
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 Return

```
class ReturnTest
{
    int value = 100;
    public void SetValue(int v) {
          value = v;
    }
    public ReturnTest Copy1() {
          Test t = new Test();
          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();
   ReturnTest t2 = t1.Copy1(); //new
   Console.WriteLine ("Copy1 t1(new) to t2");
   t1.Print("t1");      t2.Print("t2");
   t1.SetValue(50);  // 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 ();
  }
}
```

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 public void Main()
   {
      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 public void Main()
   {
      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
```

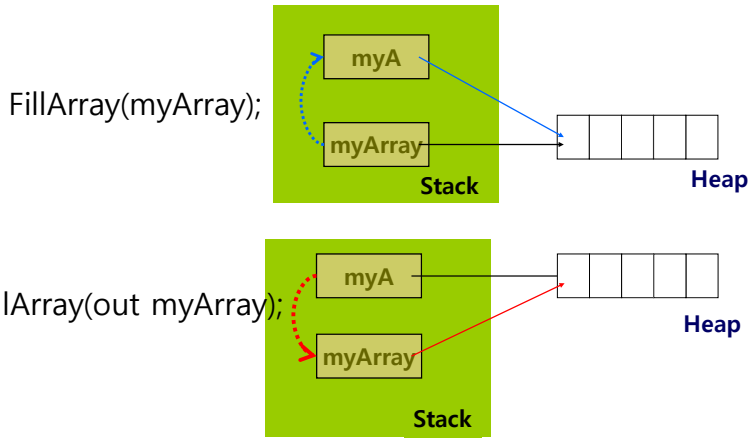
```
static void Main() {
   int x = 1;
   int y = 2;
   int result, remainder; // Initialization is not required
   Console.WriteLine(" x = {0}, y = {1}", x, y);
   Console.WriteLine("call Sum: x = {0}, y = {1},
                        sum ={2}", x, y, Sum(x,y));
   Swap(x, y);
   Console.WriteLine("call 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
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 reference
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



Method Overloading

- 메소드 오버로딩 (Method Overloading)
  - 함수의 이름은 같지만 함수의 매개변수나 반환 값이 다르게 정의
  - 호출 시 함수의 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() {
                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
        }
}
```

Overload – 다중 정의
Override – 재정의

Property

- 속성 (Property)
  - 은닉화 (encapsulation)를 위해서, private 멤버 필드에 값을 얻거나 할당하는 목적의 접근 방법을 주는 public 메소드 개념
  - 속성의 구조는 get과 set을 가지고 있음 - get만 사용하면 읽기 전용, set만 사용하면 쓰기 전용
  - set에서 사용되는 value는 매개변수 값을 디폴트로 들어오는 값으로 사용

```
private string name;
public string Name    {
    get {
        return name;  // 속성 반환 구현
    }
    set {
        name = value; // 속성 설정
    }
}
```

```
public class Car    {
        private string color;
        public string Name { // property
            get;
            set;
        }
        public string Color { // property
            get             {
                return color;
            }
            set             {
                color = value;
            }
        }
        public void Print()         {
            Console.WriteLine("Car {0}, {1}", Name, Color); // 속성이용 멤버필드 출력
        }
}
public class PropertyTest {
    static void Main() {
            Car c = new Car();
            c.Name = "Avante";  // property set
            c.Color = "White";  // property set
            Console.WriteLine("Car: {0}, {1}", c.Name, c.Color); // property get
            c.Print();          // method
    }
}
```