

클래스

UPDATE 2023

INDEX

- 객체지향프로그래밍과 클래스
- 클래스 선언과 객체

01. 객체지향 프로그래밍과 클래스

- 객체지향 프로그래밍(Object Oriented Programming) 이란?
OOP라고도 하며 코드내의 모든것을 객체(Object)로 표현하고자하는 프로그래밍 패러다임이다.
- 객체의 구성 요소 => 멤버
- 클래스안의 속성(변수, 데이터)와 메소드(기능)

01. 객체지향 프로그래밍과 클래스

- 객체의 구성 요소 => 멤버
 - 클래스안의 속성(변수, 데이터)와 메소드(기능)

객체 (Objcet)			
속성 (Property)	모양(Shape) 색상(Color) 두께(Thickness)	크기(Size) 색상(Color)	크기(Size) 색상(Color) 회전 속도(RPM)
기능 (Function)	필기(Writing) 드로잉(Drawing)	이륙(Takeoff) 비행(Flight) 착륙(Landing)	믹스(Mix)

01. 객체지향 프로그래밍과 클래스

- 클래스(Class)의 종류

- 닷넷 프레임워크에서 제공하는 내장형식(built-in type) 클래스와 사용자가 직접 클래스 구조를 만드는 사용자정의 형식(User defined type) 클래스가 있다.

“클래스는 개체를 생성하는 틀(템플릿)입니다.”

“클래스는 무엇인가를 만들어 내는 설계도입니다.”

01. 객체지향 프로그래밍과 클래스



재료



붕어빵 틀



붕어빵



객체 생성 지시



클래스(Class)



객체(Object) 생성

02. 클래스 선언과 객체 생성

- 클래스명은 반드시 대문자로 시작한다.
- 접근제어자를 생략하면 public이며 public 키워드가 붙은 클래스는 클래스 외부에서 해당 클래스를 바로 호출해서 사용할 수 있도록 공개되었다는 것을 의미한다.
- public 키워드와 반대는 private 키워드이다.

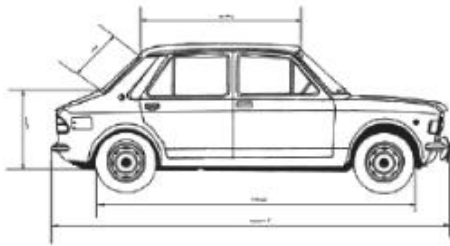
```
// 클래스 선언
접근한정자 class 클래스명
{
    // 멤버 데이터(속성. 필드) 선언
    // 메소드 구현
}
```

02. 클래스 선언과 객체 생성

- 클래스명() 은 생성자(Constructor)라고 하는 특별한 메소드로 클래스의 이름과 동일한 이름을 가지며 객체를 생성하는 역할을 한다.
- new는 생성자를 호출해서 객체를 생성하는데 사용하는 연산자이다.
- 생성된 인스턴스는 인스턴스명.속성(변수)으로 접근하여 리터럴값 할당이 가능하다.

```
// 객체 생성  
클래스명 인스턴스명 = new 클래스명();  
// 데이터 값 설정  
인스턴스명.필드명 = 값;
```


02. 클래스 선언과 객체 생성



```
class 자동차
{
    // 자동차의 특징(필드)
    자동차_색상;
    자동차_속도;
    // 자동차의 기능(메서드)
    속도_올리기( );
    속도_내리기( );
}
```

02. 클래스 선언과 객체 생성

자동차 설계도(클래스)

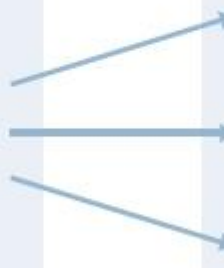
```
class 자동차
{
    // 자동차의 특징(필드)
    자동차_색상;
    자동차_속도;
    // 자동차의 기능(메서드)
    속도_올리기( );
    속도_내리기( );
}
```

자동차(인스턴스)

자동차1 = 자동차()

자동차2 = 자동차()

자동차3 = 자동차()



02. 클래스 선언과 객체 생성

단계	작업	형식	예
1단계	클래스 선언	<pre>class 클래스_이름 { // 필드 선언 // 메서드 선언 }</pre>	<pre>class Car { public String color; public void UpSpeed () { // ~~~ } }</pre>
↓			
2단계	인스턴스 생성	<pre>클래스_이름 인스턴스_이름; 인스턴스_이름 = new 클래스_이름();</pre>	<pre>Car myCar1; myCar1 = new Car();</pre>
↓			
3단계	필드나 메서드를 사용	<pre>인스턴스_이름.필드_이름 = 값; 인스턴스_이름.메서드_이름();</pre>	<pre>myCar1.color = "빨강"; myCar1.UpSpeed();</pre>

02. 클래스 선언과 객체 생성

```
class Car
{
    // 자동차의 필드
    public string color;
    public int speed;
    // 자동차의 메서드
    public void UpSpeed(int value)
    {
        speed = speed + value;
    }
    public void DownSpeed(int value)
    {
        speed = speed - value;
    }
}
```

```
Car myCar1 = new Car( );
Car myCar2 = new Car( );
Car myCar3 = new Car( );
```

```
myCar1.color = "빨간색";
myCar1.speed = 0;
myCar2.color = "파란색";
myCar2.speed = 0;
myCar3.color = "노란색";
myCar3.speed = 0;
```

02. 클래스 선언과 객체 생성

```
Console.WriteLine("자동차1의 색상은 {0}이며, 현재 속도는 {1}km 입니다.", myCar1.  
color, myCar1.speed);  
  
myCar2.UpSpeed(60);  
Console.WriteLine("자동차2의 색상은 {0}이며, 현재 속도는 {1}km 입니다.", myCar2.  
color, myCar2.speed);  
  
myCar3.UpSpeed(0);  
Console.WriteLine("자동차3의 색상은 {0}이며, 현재 속도는 {1}km 입니다.", myCar3.  
color, myCar3.speed);
```

출력 결과

자동차1의 색상은 빨강이며, 현재속도는 30km 입니다.
자동차2의 색상은 파랑이며, 현재속도는 60km 입니다.
자동차3의 색상은 노랑이며, 현재속도는 0km 입니다.

02. 클래스 선언과 객체 생성

```
// 고양이 클래스 선언
// 고양이 이름, 성별, 나이, 털색상 => 속성(필드)
// 달린다. 걷다. 먹다. => 메서드(기능)
참조 4개
class Cat
{
    // 속성(필드)
    // 접근제한자(public/private...) 데이터형 필드명;
    public string Name;
    public string Gender;
    public int Age;
    public string Color;
```

02. 클래스 선언과 객체 생성

```
//메서드(기능)  
//접근제한자(public/private...) 반환형 | void 메서드명(자료형 매개변수)
```

참조 2개

```
public void Eat()  
{  
    Console.WriteLine($"{Name} 이(가) 먹는다");  
}
```

참조 2개

```
public void Run()  
{  
    Console.WriteLine($"{Name} 이(가) 달린다.");  
}
```

```
}
```

02. 클래스 선언과 객체 생성

참조 0개

```
static void Main(string[] args)
{
    Console.WriteLine("고양이 객체 생성");

    // 클래스명 인스턴스명 = new 클래스명();
    Cat cat1 = new Cat();

    // cat1 객체에게 실제 데이터 부여
    // 인스턴스명.필드명 = 값;
    cat1.Name = "코코";
    cat1.Age = 1;
    cat1.Gender = "여자";
    cat1.Color = "흰색";
}
```


02. 클래스 선언과 객체 생성

```
// 출력
Console.WriteLine($"cat1 이름: {cat1.Name}");
Console.WriteLine($"cat1 나이: {cat1.Age}");
Console.WriteLine($"cat1 성별: {cat1.Gender}");
Console.WriteLine($"cat1 털색상: {cat1.Color}");
Console.WriteLine("=====");
```

```
고양이 객체 생성
cat1 이름: 코코
cat1 나이: 1
cat1 성별: 여자
cat1 털색상: 흰색
=====
```

02. 클래스 선언과 객체 생성

```
// 두번째 고양이 객체화
Cat cat2 = new Cat();
cat2.Name = "윈스턴";
cat2.Age = 3;
cat2.Gender = "남자";
cat2.Color = "갈색";
```

```
// 출력
Console.WriteLine($"cat2 이름: {cat2.Name}");
Console.WriteLine($"cat2 나이: {cat2.Age}");
Console.WriteLine($"cat2 성별: {cat2.Gender}");
Console.WriteLine($"cat2 털색상: {cat2.Color}");
Console.WriteLine("=====");
```

```
cat2 이름: 윈스턴
cat2 나이: 3
cat2 성별: 남자
cat2 털색상: 갈색
=====
```

02. 클래스 선언과 객체 생성

```
// 메서드 호출  
// 인스턴스명.메서드명()  
cat1.Eat();  
cat2.Eat();  
cat1.Run();  
cat2.Run();
```

코코 이(가) 먹는다
원스톤 이(가) 먹는다
코코 이(가) 달린다.
원스톤 이(가) 달린다.

퀴즈

- 타원 도형을 클래스를 이용하여 필드(이름, 반지름)와 출력과 타원 넓이를 구하는 메서드를 정의하여라
- 원의 넓이 메서드는 반환값이 있는 형태로 출력 결과를 참조한다

```
Name : Green  
Radius : 1.5  
원의 넓이 = 7.065  
원의 넓이 :  $1.5 \times 1.5 \times 3.14 = 7.065$ 
```

퀴즈

- 타원 클래스 예시

```
class Circle
{
    public string Name;
    public double Radius;

    참조 0개
    public void PrintInfo()
    {
    }
}
```

타원 정보 출력 메서드 작성

퀴즈

참조 0개

```
public double GetArea1()  
{  
}  
}
```

타원의 넓이 구하기 메서드 작성

참조 0개

```
public string GetArea2()  
{  
}  
}
```

타원의 넓이 구하기 메서드 작성

}

퀴즈

```
// Circle 클래스를 이용하여 인스턴스 생성  
Circle circle = new Circle();
```

인스턴스에 실제 값 지정

```
// 메서드 호출
```

출력 메서드와 넓이 구하기 메서드 호출

```
Name : Green  
Radius : 1.5  
원의 넓이 = 7.065  
원의 넓이 : 1.5 X 1.5 X 3.14 = 7.065
```

03. 생성자(Constructor)

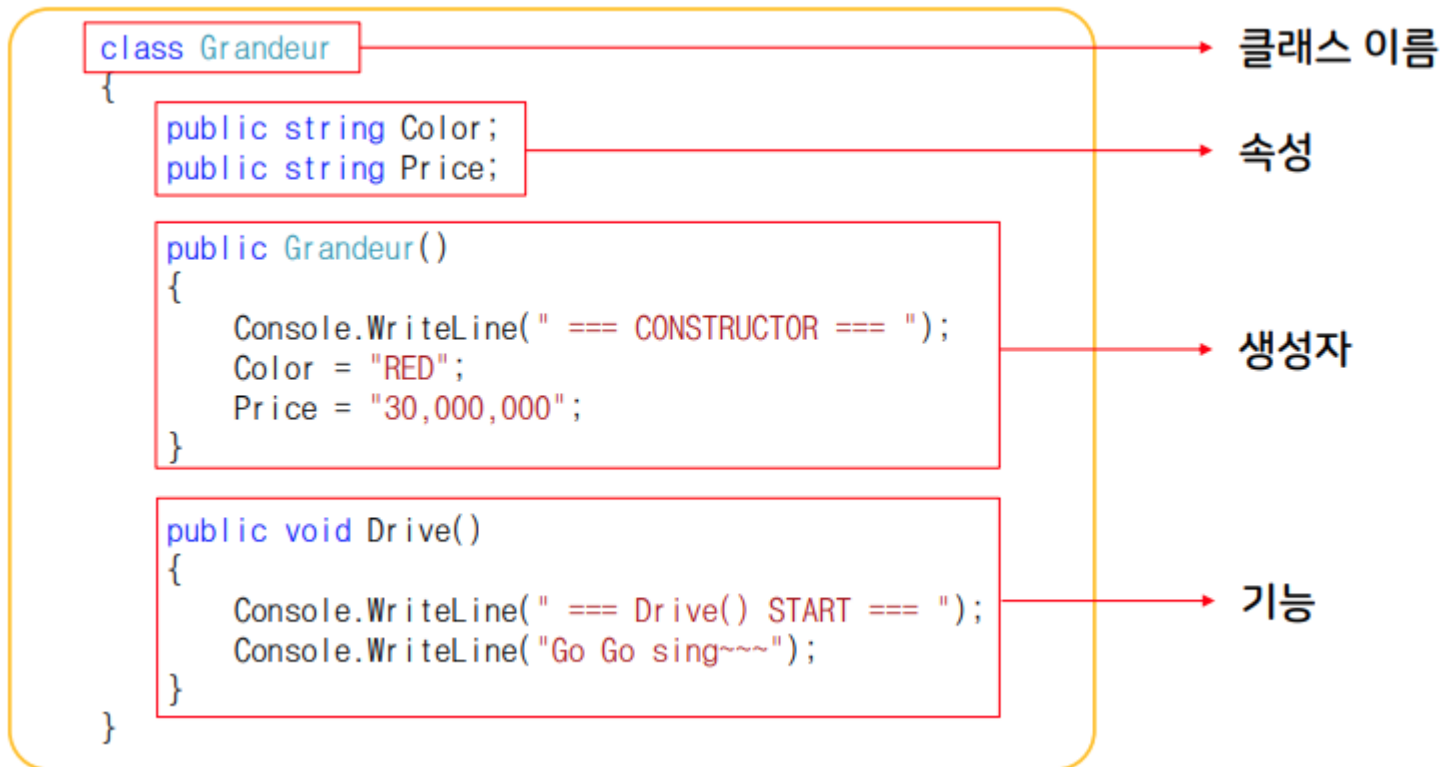
- 기본 생성자

클래스를 선언할 때 명시적으로 생성자를 구현하지 않아도 컴파일러에서 자동으로 생성되는 생성자

- 사용자정의 생성자

기본 생성자와 달리 클래스를 선언할 때 별도로 선언하는 생성자이다. 객체의 필드(변수,속성)값의 초기화 및 오버로딩이 가능하다.

03. 생성자(Constructor)



03. 생성자(Constructor)

- 필드(속성)에 값을 주는 용도로 사용하는 특별한 메서드
- return문 사용 불가.
void|반환형 이 없다.
- 메서드명이 클래스명과 동일하다

// 클래스 선언

class 클래스명

{

 // 멤버 데이터(속성. 필드) 선언

 // 생성자

public 클래스명 (데이터형 매개변수)

{ this.필드 = 매개변수; }

 // 메소드 구현

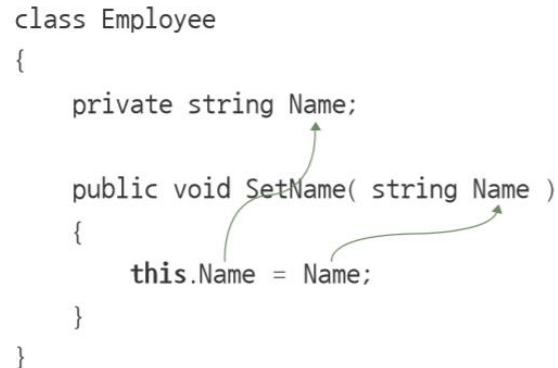
}

03. 생성자(Constructor)

- this 키워드 : 자신을 지칭하는 키워드로 자신의 필드나 메소드에 접근할 때 사용한다.

```
class Employee
{
    private string Name;

    public void SetName( string Name )
    {
        this.Name = Name;
    }
}
```

A diagram with two green arrows. One arrow starts from the 'this' in 'this.Name' and points to the 'Name' field declaration. The other arrow starts from the 'Name' parameter in the 'SetName' method signature and points to the 'Name' in 'this.Name'.

```
class MyClass
{
    int a, b, c;

    public MyClass()
    {
        this.a = 5425;
    }

    public MyClass(int b)
    {
        this.a = 5425;
        this.b = b;
    }

    public MyClass(int b, int c)
    {
        this.a = 5425;
        this.b = b;
        this.c = c;
    }
}
```

03. 생성자(Constructor)

// Dog 클래스 선언

참조 3개

class Dog

{

public string Kind;

public string Name;

참조 1개

public Dog(string kind, string name)

{

this.Kind = kind;

this.Name = name;

}

public void Bark()

{

Console.WriteLine(\$"{this.Name} : 멍멍");

}

}

03. 생성자(Constructor)

// 생성자를 이용한 데이터 값 설정

클래스명 인스턴스명 = new 클래스명(값1, 값2....);

```
Console.WriteLine("\n\n=====");  
// 생성자에 값을 전달해서 객체화  
// Dog dog1 = new Dog(); // 에러발생  
Dog dog1 = new Dog("시츄", "토토");  
Dog dog2 = new Dog("진돗개", "진돌");  
Console.WriteLine($"dog1 이름: {dog1.Name}");  
Console.WriteLine($"dog1 종류: {dog1.Kind}");  
Console.WriteLine($"dog2 이름: {dog2.Name}");  
Console.WriteLine($"dog2 종류: {dog2.Kind}");  
dog1.Bark();  
dog2.Bark();
```

```
dog1 이름: 토토  
dog1 종류: 시츄  
dog2 이름: 진돌  
dog2 종류: 진돗개  
토토 : 멍멍  
진돌 : 멍멍
```

03. 생성자(Constructor)

```
class Car
{
    public string Name;
    public string Color;
    public double Speed;

    참조 1개
    public Car(string name, string color, double speed)
    {
        this.Name = name;
        this.Color = color;
        this.Speed = speed;
    }
}
```

03. 생성자(Constructor)

참조 2개

```
public double SpeedUp(double speed)
{
    this.Speed += speed;
    return this.Speed;
}
```

참조 1개

```
public double SpeedDown(double speed)
{
    this.Speed -= speed;
    return this.Speed;
}
```

```
}
```

03. 생성자(Constructor)

```
Console.WriteLine("=====");
Car car = new Car("제네시스", "블랙", 0);
Console.WriteLine("\t 차종 : {0}, 색상 : {1}, 현재 속도 : {2}",
                  car.Name, car.Color, car.Speed);

car.SpeedUp(10);
Console.WriteLine($" \t 속도를 올립니다. 현재 속도 : {car.Speed}");
car.SpeedUp(12.5);
Console.WriteLine($" \t 속도를 올립니다. 현재 속도 : {car.Speed}");
car.SpeedDown(5.7);
Console.WriteLine($" \t 속도를 줄입니다. 현재 속도 : {car.Speed}");
```

```
차종 : 제네시스, 색상 : 블랙, 현재 속도 : 0
속도를 올립니다. 현재 속도 : 10
속도를 올립니다. 현재 속도 : 22.5
속도를 줄입니다. 현재 속도 : 16.8
```


퀴즈

- 타원 도형을 클래스를 이용하여 필드(이름, 반지름)와 출력과 타원 넓이를 구하는 메서드를 정의하여라
- 원의 넓이 메서드는 반환값이 있는 형태로 출력 결과를 참조한다

```
Name : Green  
Radius : 1.5  
원의 넓이 = 7.065  
원의 넓이 : 1.5 X 1.5 X 3.14 = 7.065
```

퀴즈

- 클래스를 이용하여 도서관리 데이터를 관리하도록 프로그래밍 하여라.
- 도서관리 데이터의 필드는 ISBN 번호, 상품명, 작가, 정가로 한다.
- 생성자 메서드를 이용하여야 한다.
- 클래스 선언시 도서 정보를 출력하는 메서드를 별도로 정의하여 호출하도록 한다
- 도서 할인율은 전달값에 따라 도서 정가를 이용하여 할인가를 구하여 출력하도록 한다.

퀴즈

첫번째 책 정보 : 할인율 0.25

ISBN : 9788932917245

제 목 : 어린왕자

저 자 : 생텍쥐페리

정 가 : 15,000 원

할인가(25%) : 11,250 원

두번째 책 정보 : 할인율 0.3

ISBN : 9791162243770

제 목 : 이것이 C#이다

저 자 : 박상현

정 가 : 35,000 원

할인가(30%) : 24,500 원

퀴즈

```
class Book
```

```
{
```

필드 선언

참조 2개

```
public Book(long isbn, string title, string writer, double price)
```

```
{
```

생성자 메서드

```
}
```

퀴즈

```
public void Print_book(double Rate)
```

```
{
```

도서 정보 출력 메서드, Rate는 도서할인율
할인가=정가 - (정가*할인율)

```
}
```

```
;
```

```
}
```

퀴즈

// 책 인스턴스 생성

인스턴스 생성 및 책 데이터 값 전달

// 출력

```
Console.WriteLine("\n=====");
```

```
var rate = 0.25;
```

```
Console.WriteLine($"첫번째 책 정보 : 할인율 {rate} ");
```

출력 메서드를 이용하여 출력1

```
Console.WriteLine();
```

```
rate = 0.3;
```

```
Console.WriteLine($"두번째 책 정보 : 할인율 {rate} ");
```

출력 메서드를 이용하여 출력2

04. 메서드 오버로딩

- 메서드 오버로딩(Overloading) : 같은 클래스 내에서 메서드의 이름이 같아도 파라미터의 매개변수 갯수나 데이터 타입만 다른 선언을 여러개 할수 있는것을 말한다.
- 클래스안에서는 생성자 메서드 오버로딩과 일반 메서드 오버로딩이 있다.

04. 오버로딩

- 생성자 메서드 오버로딩
메서드 이름은 클래스명으로 동일
하지만 매개변수의 자료형과 개수
가 다른 경우이다.

```
Car myCar1 = new Car();  
Car myCar2 = new Car("빨강");  
Car myCar3 = new Car("파랑", 30);
```

```
public Car()  
{  
}  
  
public Car(string color)  
{  
    this.color = color;  
}  
  
public Car(string color, int speed)  
{  
    this.color = color;  
    this.speed = speed;  
}
```


04. 메서드 오버로딩

```
class Car
```

```
{
```

```
    public string color;  
    public int speed;
```

참조 1개

```
    public Car()
```

```
    {  
    }
```

참조 1개

```
    public Car(string color)
```

```
    {  
        this.color = color;  
    }
```

참조 1개

```
    public Car(string color, int speed)
```

```
    {  
        this.color = color;  
        this.speed = speed;  
    }
```

참조 3개

```
    public string GetColor()
```

```
    {  
        return color;  
    }
```

참조 3개

```
    public int GetSpeed()
```

```
    {  
        return speed;  
    }
```

```
}
```

04. 메서드 오버로딩

```
Console.WriteLine("\t생성자 메서드 오버로딩 테스트");  
Car myCar1 = new Car();  
Car myCar2 = new Car("빨강");  
Car myCar3 = new Car("파랑", 30);
```

```
Console.WriteLine("자동차1의 색상은 {0}이며, 현재속도는 {1}km 입니다.",  
    myCar1.GetColor(), myCar1.GetSpeed());  
Console.WriteLine("자동차2의 색상은 {0}이며, 현재속도는 {1}km 입니다.",  
    myCar2.GetColor(), myCar2.GetSpeed());  
Console.WriteLine("자동차3의 색상은 {0}이며, 현재속도는 {1}km 입니다.",  
    myCar3.GetColor(), myCar3.GetSpeed());
```

생성자 메서드 오버로딩 테스트
자동차1의 색상은 이며, 현재속도는 0km 입니다.
자동차2의 색상은 빨강이며, 현재속도는 0km 입니다.
자동차3의 색상은 파랑이며, 현재속도는 30km 입니다.

04. 오버로딩

- 일반 메서드 오버로딩

메서드 이름은 동일하지만 매개변수의 데이터 형과 개수가 틀린 경우이다.

```
// 일반 메서드 오버로딩 테스트
// 매개변수의 데이터형이 틀리다.
참조 2개
class Calc
{
    참조 1개
    public void addValue(double v1, double v2)
    {
        Console.WriteLine("double값 계산 ==> {0}", (v1 + v2));
    }
    참조 1개
    public void addValue(int v1, int v2)
    {
        Console.WriteLine("int값 계산 ==> {0}", (v1 + v2));
    }
}
```

04. 오버로딩

```
Console.WriteLine("\n=====");  
Console.WriteLine("\t일반 메서드 오버로딩 테스트");  
Calc myCalc = new Calc();  
myCalc.addValue(1.1, 1.1);  
myCalc.addValue(1, 1);
```

일반 메서드 오버로딩 테스트
double값 계산 ==> 2.2
int값 계산 ==> 2

04. 오버로딩

오버로딩이 성립하기 위한 조건 (3가지)

1. 메서드 이름이 같아야 한다.
2. 매개변수의 개수 또는 타입이 달라야 한다.
3. 반환 타입은 영향없다.

퀴즈

- Figure 클래스를 선언하여 생성자 메서드 오버로딩 방식을 이용하여 타원, 사각형, 사다리꼴의 데이터 값에 따라 각 도형의 정보와 넓이를 출력하도록 프로그래밍 하여라.

// Main 메서드 코딩 예시

```
Figure circle = new Figure(2.5);  
Figure square = new Figure(4.5, 3.0);  
Figure trapezoid = new Figure(1.5, 3.0, 2.5);  
circle.PrintInfo();  
square.PrintInfo();  
trapezoid.PrintInfo();
```

입력 및 출력 오류

=====

타원 도형 정보

반지름 : 2.5 cm

넓이 : 19.625 cm

=====

사각형 도형 정보

가로 : 4.5 cm

높이 : 3 cm

넓이 : 13.5 cm

=====

사다리꼴 도형 정보

윗변 : 1.5 cm

아랫변 : 3 cm

높이 : 2.5 cm

넓이 : 5.625 cm

05. 정적 필드와 정적 메서드

- `static` 키워드를 이용하여 메소드나 필드가 클래스의 인스턴스가 아닌 클래스 자체에 소속되도록 지정하는 접근 한정자이다.
- 클래스가 여러 개 있는 경우 프로그램 전체에 공유하는 변수나 공유 메소드로 사용한다.
- 호출할 경우 인스턴스를 만들지 않고 클래스명.변수로 직접 접근할 수 있다.

퀴즈

// Figure 클래스 코딩 예시

```
class Figure
{
    public double X, Y, Z;
```

생성자 메서드 선언

```
public void PrintInfo()
{
    Console.WriteLine("=====");
```

조건문 등을 이용하여
도형의 정보 출력 및
도형의 넓이 출력 메서드 정의

```
}
}
```


05. 정적 필드와 정적 메서드

- `static` 키워드를 이용하여 메소드나 필드가 클래스의 인스턴스가 아닌 클래스 자체에 소속되도록 지정하는 접근 한정자이다.
- 클래스가 여러 개 있는 경우 프로그램 전체에 공유하는 변수나 공유 메소드로 사용한다.
- 호출할 경우 인스턴스를 만들지 않고 클래스명.변수로 직접 접근할 수 있다

05. 정적 필드와 정적 메서드

```
클래스명()
{
    public static 자료형 변수명;

    public static 자료형 메서드명(매개변수)
    {
        // 명령문
    }
}
```

```
클래스명.변수 = 리터럴값;
```

```
클래스명.메서드명(값);
```

05. 정적 필드와 정적 메서드

인스턴스에 소속된 필드의 경우	클래스에 소속된 필드의 경우(static)
<pre>class MyClass { public int a; public int b; } // public static void Main() { MyClass obj1 = new MyClass(); obj1.a = 1; obj1.b = 2; MyClass obj2 = new MyClass();</pre>	<pre>class MyClass { public static int a; public static int b; } // public static void Main() { MyClass.a = 1; MyClass.b = 2; }</pre> <div data-bbox="1188 1096 1657 1220"><p>인스턴스를 만들지 않고 클래스의 이름을 통해 필드에 직접 접근합니다.</p></div>

05. 정적 필드와 정적 메서드

```
class MyClass
{
    public static int A;           // 정적 필드
    public static int B;

    public static void Print_number1() // 정적 메서드
    {
        Console.WriteLine($" {A} , {B} ");
    }

    참조 1개
    public void Print_number2()      // 일반 메서드
    {
        Console.WriteLine($" {A} , {B} ");
    }
} // End MyClass
```

05. 정적 필드와 정적 메서드

```
Console.WriteLine("\n=====");
Console.WriteLine("\t정적 필드와 정적 메서드 테스트");
// 정적 필드에 값 지정 후 정적 메서드 호출
MyClass.A = 100;
MyClass.B = 100;
MyClass.Print_number1();

// 인스턴스 생성 후 인스턴스 메서드 호출
MyClass myclass = new MyClass();
myclass.Print_number2();
```

```
정적 필드와 정적 메서드 테스트
100 , 100
100 , 100
```

05. 정적 필드와 정적 메서드

참조 5개

```
class Bread
```

```
{
```

```
    // 정적 필드와 일반 필드를 함께 선언
```

```
    public static string Brand = "파리바게뜨";
```

```
    public string Kind; // 빵종류
```

```
    public decimal Price; // 빵 가격
```

```
    public double Kcal; // 칼로리
```

```
    // 정적 필드를 사용한 정적 메서드
```

참조 1개

```
    public static void Info()
```

```
{
```

```
    // Console.WriteLine(this.Brand); // 오류 발생
```

```
    Console.WriteLine("\t\t" + Brand);
```

```
    Console.WriteLine("\t\t주소 : 부산시 금정구 장전동 ... ");
```

```
}
```

05. 정적 필드와 정적 메서드

// 생성자 메서드

참조 1개

```
public Bread(string kind, decimal price, double kcal)
{
    this.Kind = kind;
    this.Price = price;
    this.Kcal = kcal;
}
```

// 일반 메서드

참조 1개

```
public void Bread_info()
{
    Console.WriteLine();
    Console.WriteLine($"종류 : {this.Kind}");
    Console.WriteLine($"가격 : {this.Price} 원 ");
    Console.WriteLine($"칼로리 : {this.Kcal} kcal");
}
}
```

05. 정적 필드와 정적 메서드

```
// 정적 메서드와 정적 필드 호출  
Bread.Info();  
Console.WriteLine("\n\t\t" + Bread.Brand);  
Console.WriteLine("\n\n");
```

```
// 객체 생성 및 객체 메서드 호출  
Bread baguette = new Bread("바게뜨", 4_500, 250);  
baguette.Bread_info();
```

파리바게뜨
주소 : 부산시 금정구 장전동 ...
파리바게뜨

종류 : 바게뜨
가격 : 4500 원
칼로리 : 250 kcal

05. 정적 필드와 정적 메서드

// 서로 다른 클래스에서 정적 필드 사용

참조 5개

```
class Global
{
    // 정적 필드
    public static int Count = 0;

    // 정적 메서드
    참조 1개
    public static void method_static()
    {
        Console.WriteLine("정적 메서드 호출");
    }
}
```

class ClassA

```
{
    참조 2개
    public ClassA()
    {
        Global.Count++;
    }
}
```

참조 3개

```
class ClassB
{
    참조 2개
    public ClassB()
    {
        Global.Count++;
    }
}
```

05. 정적 필드와 정적 메서드

```
Console.WriteLine($"Global Count => {Global.Count}");  
Global.method_static();
```

```
// 클래스 생성자 호출
```

```
new ClassA();  
new ClassA();  
new ClassB();  
new ClassB();  
Console.WriteLine($"Global Count => {Global.Count}");
```

```
Global Count => 0  
정적 메소드 호출  
Global Count => 4
```

퀴즈

- 출생년도 데이터를 입력하면 12간지 띠를 구하는 클래스와 메서드를 작성하여라.
- 메서드는 정적 메서드를 이용한다.
- 알고리즘 : 출생년 % 12 = ?
나머지 값에 따라 띠가 결정된다.

// 띠 순서

"원숭이", "닭", "개", "돼지", "쥐", "소", "호랑이", "토끼",
"용", "뱀", "말", "양"

2009년 생은 소 띠
2023년 생은 토끼 띠
1997년 생은 토끼 띠

퀴즈

```
class MyZodiac
```

```
{
```

참조 1개

```
public static string GetZodiac(int BirthYear)
```

```
{
```

// 띠 구하기 프로그래밍 소스 입력

```
}
```

```
}
```

```
// 정적 메서드를 출력문 안에서 호출
```

```
Console.WriteLine($"{\t{2009}년 생은 { 정적 메서드 호출 } 띠 ");
```

```
Console.WriteLine($"{\t{2023}년 생은 { 정적 메서드 호출 } 띠 ");
```

```
Console.WriteLine($"{\t{1997}년 생은 { 정적 메서드 호출 } 띠 ");
```

06. 객체 복사

- 객체 복사(object copying)는 객체 지향 프로그램에서 말하는 데이터 단위인 기존의 객체의 사본을 생성하는 것이다.
- 객체 사본"(object copy) 또는 "사본"(copy) 이라고 한다
- 복사방식에는 얇은 복사와 깊은 복사가 있다.
- 위키 백과

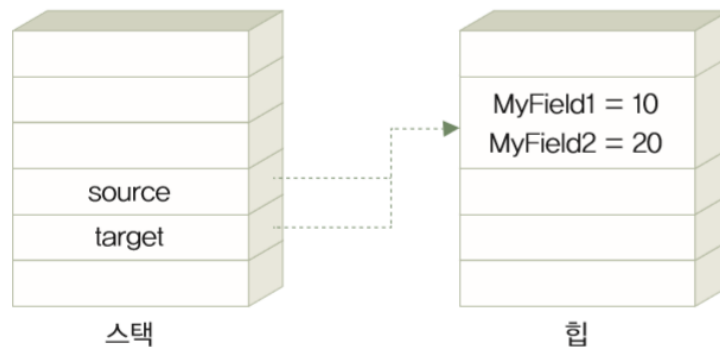
https://ko.wikipedia.org/wiki/%EA%B0%9D%EC%B2%B4_%EB%B3%B5%EC%82%AC

06. 객체 복사

- 얕은 복사 (Shallow Copy)

객체를 복사할때 참조 복사 방식으로 복사한다.

객체가 같은 메모리 영역을 사용하므로 객체 중 하나의 값이 변경되면 함께 변경된다.

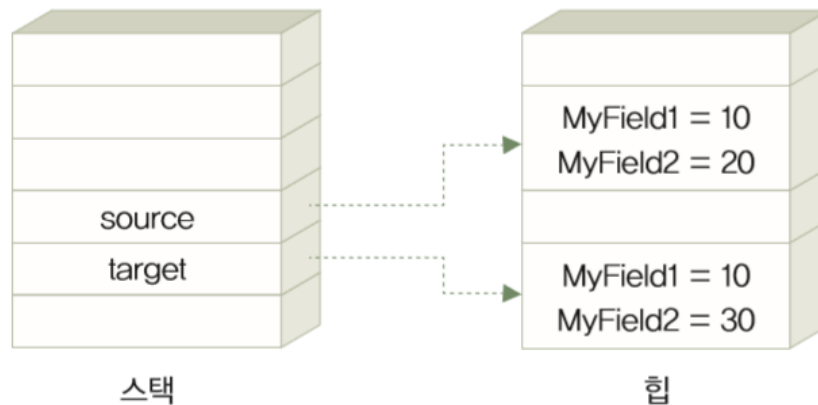


06. 객체 복사

- 깊은 복사 (Deep Copy)

객체를 복사할때 값 복사 방식으로 복사한다.

객체가 서로 다른 메모리 영역을 사용하므로 객체 중 하나의 값이 변경되어도 다른 객체에는 영향력이 없다



06. 객체 복사

```
class MyClass
{
    public int MyField1;
    public int MyField2;
}

// 얕은 복사 테스트;
MyClass source = new MyClass();
source.MyField1 = 10;
source.MyField2 = 20;

// 새 객체인스턴스를 source 객체인스턴스에서 복사
MyClass target = source;

// 출력
Console.WriteLine($"source.MyField1 => {source.MyField1}"); //10
Console.WriteLine($"source.MyField2 => {source.MyField2}"); //20
Console.WriteLine($"target.MyField1 => {target.MyField1}"); //10
Console.WriteLine($"target.MyField2 => {target.MyField2}"); //20
```


06. 객체 복사

```
Console.WriteLine("=====");  
// 복사 객체의 값 변경후 확인  
target.MyField2 = 30;  
Console.WriteLine($"source.MyField1 => {source.MyField1}"); // 10  
Console.WriteLine($"source.MyField2 => {source.MyField2}"); // 30  
Console.WriteLine($"target.MyField1 => {target.MyField1}"); // 10  
Console.WriteLine($"target.MyField2 => {target.MyField2}"); // 30
```

```
source.MyField1 => 10  
source.MyField2 => 20  
target.MyField1 => 10  
target.MyField2 => 20  
=====  
source.MyField1 => 10  
source.MyField2 => 30  
target.MyField1 => 10  
target.MyField2 => 30
```

07. 구조체 VS 클래스

특징	클래스	구조체
키워드	class	struct
형식	참조 형식	값 형식
복사	얕은 복사(Shallow Copy)	깊은 복사(Deep Copy)
인스턴스 생성	new 연산자와 생성자 필요	선언만으로도 생성
생성자	매개 변수 없는 생성자 선언 가능	매개 변수 없는 생성자 선언 불가능
상속	가능	모든 구조체는 System.Object 형식을 상속하는 System.ValueType으로부터 직접 상속받음

```
struct 구조체이름
{
    // 필드 ...
    // 메소드 ...
}
```

```
struct MyStruct
{
    public int MyField1
    public int MyFiled2

    public void MyMethod()
    {
        // ...
    }
}
```

07. 구조체 VS 클래스

// 구조체 => 값 방식

참조 3개

```
struct MyStruct
```

```
{  
  .....  
}
```

```
    public int a;  
    public int b;
```

```
MyStruct mystruct1 = new MyStruct();  
mystruct1.a = 100;  
mystruct1.b = 200;
```

```
MyStruct mystruct2 = mystruct1;  
Console.WriteLine($"mystruct1.a => {mystruct1.a}"); // 100  
Console.WriteLine($"mystruct1.b => {mystruct1.b}"); // 200  
Console.WriteLine($"mystruct2.a => {mystruct2.a}"); // 100  
Console.WriteLine($"mystruct2.b => {mystruct2.b}"); // 200  
Console.WriteLine("\n\n =====");
```

```
mystruct1.a => 100  
mystruct1.b => 200  
mystruct2.a => 100  
mystruct2.b => 200
```

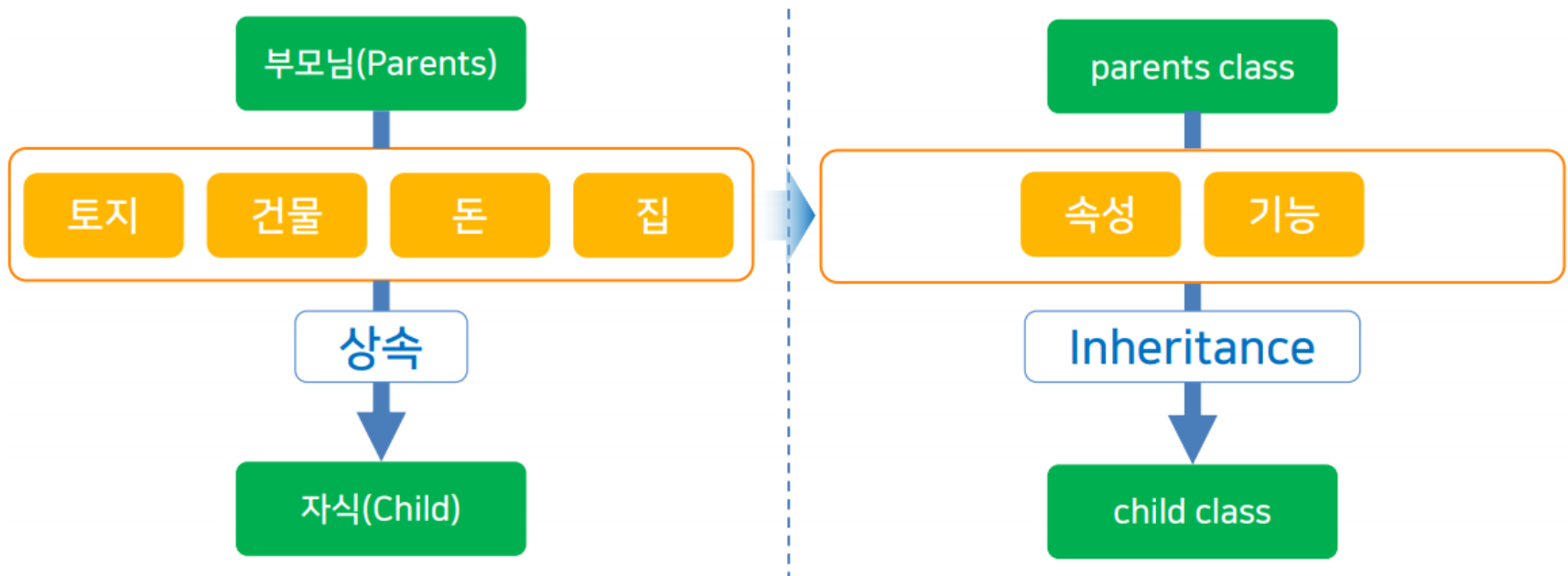
07. 구조체 VS 클래스

```
mystruct2.a = 500;  
Console.WriteLine($"mystruct1.a => {mystruct1.a}"); // 100  
Console.WriteLine($"mystruct1.b => {mystruct1.b}"); // 200  
Console.WriteLine($"mystruct2.a => {mystruct2.a}"); // 500  
Console.WriteLine($"mystruct2.b => {mystruct2.b}"); // 200
```

```
=====  
mystruct1.a => 100  
mystruct1.b => 200  
mystruct2.a => 500  
mystruct2.b => 200
```

08. 상속(Inheritance)

상속은 부모님이 자식한테 토지, 건물등을 상속하는 것과 같이 클래스에서 속성과 메서드를 상속하는 기능이다.



08. 상속(Inheritance)

- 상속 관계에서 자식 클래스를 파생 클래스(Derived Class), 서브 클래스라고 하며 부모 클래스를 기반 클래스(Base Class), 슈퍼 클래스라고 한다.

Parents class
Super class
Base class



Child class
Sub class
Derived class

파생 클래스

새로운 멤버

기반 클래스

08. 상속(Inheritance)

- 클래스 상속은 단일 상속(single inheritance)과 다중 상속(multiple inheritance)으로 구분할 수 있지만 C#의 클래스 상속은 단일 상속만 지원하며 다중 상속은 인터페이스 기능을 이용해야만 가능하다.
- 클래스 상속 구문 클래스명 뒤에 콜론(:)과 부모클래스명을 기입한다.

```
class Parents  
{  
    ...  
}
```

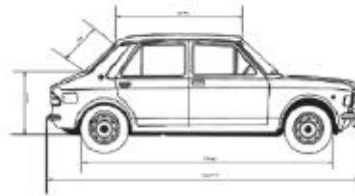


```
class Child : Parents  
{  
    ...  
}
```

08. 상속(Inheritance)

```
class 자동차  
{  
    필드 : 색상, 속도  
    메서드 : 속도_올리기()  
            속도_내리기()  
}
```

자동차 클래스(공통 내용)



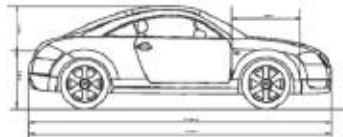
상속



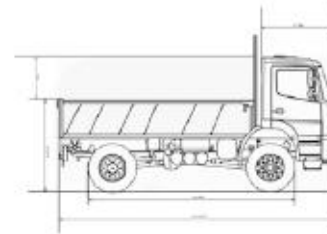
상속



승용차 클래스



트럭 클래스



08. 상속(Inheritance)

```
class 승용차 : 자동차를 상속
{
    필드 : 자동차 필드
           + 좌석수
    메서드 : 자동차 메서드
           + 좌석수_알아보기()
}
```

```
class 트럭 : 자동차를 상속
{
    필드 : 자동차 필드
           + 적재량
    메서드 : 자동차 메서드
           + 적재량_알아보기()
}
```

08. 상속(Inheritance)

```
class Parent
{
    참조 1개
    public void Foo()
    {
        Console.WriteLine("부모 메소드");
    }
}
```

// 상속받은 Child 클래스

```
참조 2개
class Child : Parent
{
    참조 1개
    public void Bar()
    {
        Console.WriteLine("자식 메소드");
    }
}
```

```
// 자식클래스의 인스턴스 생성
Child child = new Child();
child.Foo();
child.Bar();
```

부모 메소드
자식 메소드

08. 상속(Inheritance)

```
class Animal
{
    public int Leg = 4;

    참조 2개
    public void Eat()
    {
        Console.WriteLine("===== 먹다 =====");
    }
}

class Tiger : Animal
{
    public string Gender;
    public string Name;

    참조 1개
    public void Run()
    {
        Console.WriteLine("===== 달리다 =====");
    }
}
```

08. 상속(Inheritance)

```
// 부모 클래스 인스턴스
Animal animal = new Animal();
Console.WriteLine(animal.Leg);
animal.Eat();

// 자식 클래스 인스턴스
Tiger tiger = new Tiger();
Console.WriteLine(tiger.Leg);
tiger.Name = "호순이";
tiger.Gender = "여";
Console.WriteLine(tiger.Name);
Console.WriteLine(tiger.Gender);
tiger.Eat();
tiger.Run();
```

```
4
===== 먹다 =====
```

```
4
호순이
여
===== 먹다 =====
===== 달리다 =====
```

08. 상속(Inheritance)

```
public class Car
{
    public string color;
    public int speed;

    참조 0개
    public void UpSpeed(int value)
    {
        speed = speed + value;
    }

    참조 0개
    public void DownSpeed(int value)
    {
        speed = speed - value;
    }
}
```

08. 상속(Inheritance)

```
public class Sedan : Car
{
    public int seatNum;

    참조 0개
    public int GetSeatNum()
    {
        return seatNum;
    }
}
```

```
참조 0개
public class Truck : Car
{
    public int capacity;

    참조 0개
    public int GetCapacity()
    {
        return capacity;
    }
}
```

08. 상속(Inheritance)

```
Sedan sedan = new Sedan();  
Truck truck = new Truck();
```

```
sedan.UpSpeed(150);  
truck.UpSpeed(100);
```

```
sedan.seatNum = 5;  
truck.capacity = 50;
```

```
Console.WriteLine("승용차 속도는 {0}km, 좌석수는 {1}개 입니다 ",  
    sedan.speed, sedan.GetSeatNum());  
Console.WriteLine("트럭 속도는 {0}km, 적재량은 {1}톤 입니다 ",  
    truck.speed, truck.GetCapacity());
```

승용차 속도는 150km, 좌석수는 5개 입니다
트럭 속도는 100km, 적재량은 50톤 입니다

퀴즈

- Dinosaur 부모 클래스를 상속받아 Tyrano 자식 클래스와 Dooly 자식 클래스를 선언하고 인스턴스를 만들어 실행하도록 하여라.
- Dinosaur 부모 클래스는 Kind 필드와 Eat(), Sleep() 메서드가 있다.
- Tyrano 자식 클래스는 Name 필드와 Hunt() 메서드를 선언해야한다.
- Dooly 자식 클래스는 Name 필드와 Sing(), Dance() 메서드를 선언해야한다.

```
공룡 티라노사우루스
    동굴에서 잔다!!!
    티라노사우루스가 물고기를/을 사냥한다!!!
    물고기를/을 먹는다!!!

공룡 둘리
    공원에서 잔다!!!
    치킨를/을 먹는다!!!
    둘리가 춤춘다!!!
    둘리가 노래한다!!!
```


퀴즈

```
class Dinosaur
```

```
{
```

부모 클래스의 메서드 선언

```
}
```

퀴즈

```
class Tyranno: Dinosaur
```

```
{
```

자식 클래스의 필드와 메서드 선언

```
}
```

퀴즈

```
class Dooly : Dinosaur
```

```
{
```

자식 클래스의 필드와 메서드 선언

```
}
```

퀴즈

```
Tyranno tyranno = new Tyranno();
```

인스턴스 생성 후 메서드와 인스턴스 변수 호출

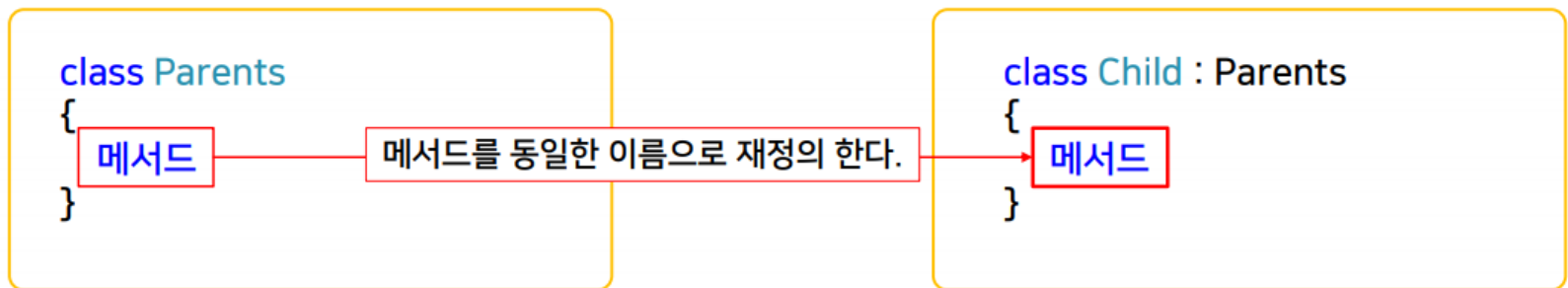
```
Console.WriteLine();
```

```
Dooly dooly = new Dooly();
```

인스턴스 생성 후 메서드와 인스턴스 변수 호출

09. 오버라이딩

- 오버라이딩(overriding)
override – 무시하다. 상위 메서드를 무시하고 하위(자식클래스)에서 재정의 하는 기능이다.
- 부모 클래스의 메서드를 자식 클래스에서 재정의 한다.



`public virtual void makeJjajang()` — 오버라이딩 —→ `public override void makeJjajang()`

09. 오버라이딩

- 오버라이드의 대상은? 클래스 메서드 , 속성(필드) , 인덱서, 이벤트...
- 상위 (부모 클래스) 대상을 virtual 키워드로 선언해야 한다.
- private로 선언한 메소드는 오버라이딩을 할 수 없다.
- 하위 클래스에서 오버라이딩을 적용할 대상은 재정의의 위해 override 키워드 사용해야한다.
- 상속한 메서드와 같은 이름, 같은 매개변수의 메소드로 선언해야한다.
반환값의 데이터형도 같아야 한다.

09. 오버라이딩

// 부모 클래스의 메서드에서 virtual 키워드 이용

참조 3개

```
class Animal
```

```
{
```

```
    public string Name = "Animal";
```

참조 3개

```
    public virtual void Eat()
```

```
    {
```

```
        Console.WriteLine("Animal : Eat");
```

```
    }
```

```
}
```

// 자식 클래스의 메서드에서 override 키워드 이용

참조 2개

```
class Cat : Animal
```

```
{
```

```
    public string Name = "Cat"; // Name 필드값 초기화
```

참조 3개

```
    public override void Eat()
```

```
    {
```

```
        Console.WriteLine("Cat : Eat");
```

```
    }
```

```
}
```

09. 오버라이딩

```
// 부모 클래스 인스턴스 생성
Animal animal = new Animal();
animal.Eat();
Console.WriteLine(animal.Name); // Animal
```

```
// 자식 클래스 인스턴스 생성
Cat cat = new Cat();
cat.Eat();
Console.WriteLine(cat.Name);
```

```
Animal : Eat
Animal
```

```
Cat : Eat
Cat
```


09. 오버라이딩

```
public class Car
{
    public int Speed;

    참조 3개
    public virtual void UpSpeed(int speed)
    {
        this.Speed += speed;
        Console.WriteLine("\t현재속도(기본 클래스) : {0}km", this.Speed);
    }
}
```

09. 오버라이딩

```
public class Sedan : Car
{
    참조 3개
    public override void UpSpeed(int speed)
    {
        this.Speed += speed;
        // 최고 속도를 150으로 제한
        if (this.Speed > 150) this.Speed = 150;
        Console.WriteLine("\t현재속도(Sedan 클래스) : {0}km", this.Speed);
    }
}
```

```
참조 2개
public class Truck : Car
{
}
```

09. 오버라이딩

```
Sedan sedan = new Sedan();  
Truck truck = new Truck();  
  
Console.WriteLine("승용차 ---> ");  
sedan.UpSpeed(200);  
  
Console.WriteLine("트럭 ---> ");  
truck.UpSpeed(200);
```

승용차 --->	현재속도(Sedan 클래스) : 150km
트럭 --->	현재속도(기본 클래스) : 200km

10. base 키워드

- base : 자식 클래스에서 부모 클래스에서 정의한 멤버 필드를 사용할 때 이용한다.
예) base.필드명
- 이름이 겹치는 등 특수한 이유로 부모의 메서드에 접근 불가할 경우 this 키워드와 같은 형태로 base 키워드를 사용한다.
(this : 자신 나타내는 키워드, base : 부모 클래스를 나타내는 키워드)

10. base 키워드

```
class Animal
{
    public string Name = "Animal";
    참조 4개
    public virtual void Eat()
    {
        Console.WriteLine("Animal : Eat");
    }
}
```

10. base 키워드

```
class Cat : Animal
{
    public string Name = "Cat"; // Name 필드값 초기화
    참조 4개
    public override void Eat()
    {
        Console.WriteLine("Cat : Eat");
    }

    // 부모 클래스의 필드와 메서드 사용
    참조 0개
    public void Who_parent()
    {
        Console.WriteLine(base.Name);
        base.Eat();
    }
}
```

10. base 키워드

```
Console.WriteLine();  
// 자식 클래스 인스턴스 생성  
Cat cat = new Cat();  
Console.WriteLine(cat.Name);  
cat.Eat();  
cat.Who_parent();
```

```
Cat  
Cat : Eat  
Animal  
Animal : Eat
```

11. 접근제어자

- 캡슐화(Encapsulation)

객체의 속성(data fields)과 행위(methods)를 하나로 묶고 실제 구현 내용 일부를 외부에 감추어 은닉한다.

감추고 싶은 것은 감추고 보여주고 싶은 것만 보여준다.

- 접근제어자(Access Modifier)

클래스에서 선언된 필드와 메서드가 외부에서 접근하지 못하도록 제한하는 역할을 한다.

객체의 캡슐화를 구현하는 기능.

접근제어자가 생략된 클래스 멤버는 private로 자동 지정된다.

11. 접근제어자

```
class MyClass
{
    private int MyField_1;
    protected int MyField_2;
    int MyField_3;

    public int MyMethod_1( )
    {
        // ...
    }

    internal void MyMethod_1 ( )
    {
        // ...
    }
}
```

접근 한정자로 수식하지 않으면
private와 같은 공개 수준을 가집니다.

[접근 제한자] [자료형] [변수 이름]
[접근 제한자] [반환형] [메서드 이름]([매개변수])
{
 [메서드 코드]
}

11. 접근제어자

- private : 같은 클래스 내부에서만 접근이 가능
- public : 모든 곳에서 해당 멤버(필드, 메서드등)로 접근이 가능
- internal : 같은 어셈블리(현재 프로젝트의 모든 클래스)에서만 public으로 접근이 가능
- protected : 클래스 외부에서 접근할 수 없으나 파생 클래스(자식클래스)에서는 접근이 가능

11. 접근제어자

```
class MyClassA
{
    // 공개용
    public int X = 100;

    // 같은 클래스에서만 허용. private 생략 가능
    private int Y = 200;

    // 같은 클래스나 상속관계의 자식 클래스에서만 허용
    protected int Z = 300;

    public int GetY()
    {
        return Y;
    }

    참조 2개
    public int GetZ()
    {
        return Z;
    }

    public void SetY(int y)
    {
        this.Y = y;
    }

    public void SetZ(int z)
    {
        this.Z = z;
    }
}
```

11. 접근제어자

참조 2개

```
class MyClassB : MyClassA
```

```
{
```

참조 2개

```
public void Get_info()
```

```
{
```

```
    Console.WriteLine($"X = {this.X}");
```

```
    // Console.WriteLine($"Y = {this.Y}");
```

```
    Console.WriteLine($"Z = {this.Z}");
```

```
}
```

```
//X, Z 필드값 변경 메서드
```

참조 1개

```
public void Set_X_Z(int x, int z)
```

```
{
```

```
    this.X = x;
```

```
    this.Z = z;
```

```
}
```

```
}
```

11. 접근제어자

```
MyClassA class_a = new MyClassA();  
Console.WriteLine($"X = {class_a.X}");  
class_a.X = 500;  
Console.WriteLine($"X = {class_a.X}");  
  
//class_a.Y = 900;  
//Console.WriteLine($"Y = {class_a.Y}");  
Console.WriteLine($"Y = {class_a.GetY()}");  
class_a.SetY(900);  
Console.WriteLine($"Y = {class_a.GetY()}");
```

```
=====  
X = 100  
X = 500  
=====  
Y = 200  
Y = 900
```

11. 접근제어자

```
MyClassB class_b = new MyClassB();  
class_b.Get_info();
```

```
//class_b.Z = 77;  
class_b.Set_X_Z(55, 77);  
class_b.Get_info();
```

```
X = 100  
Z = 300  
X = 55  
Z = 77
```