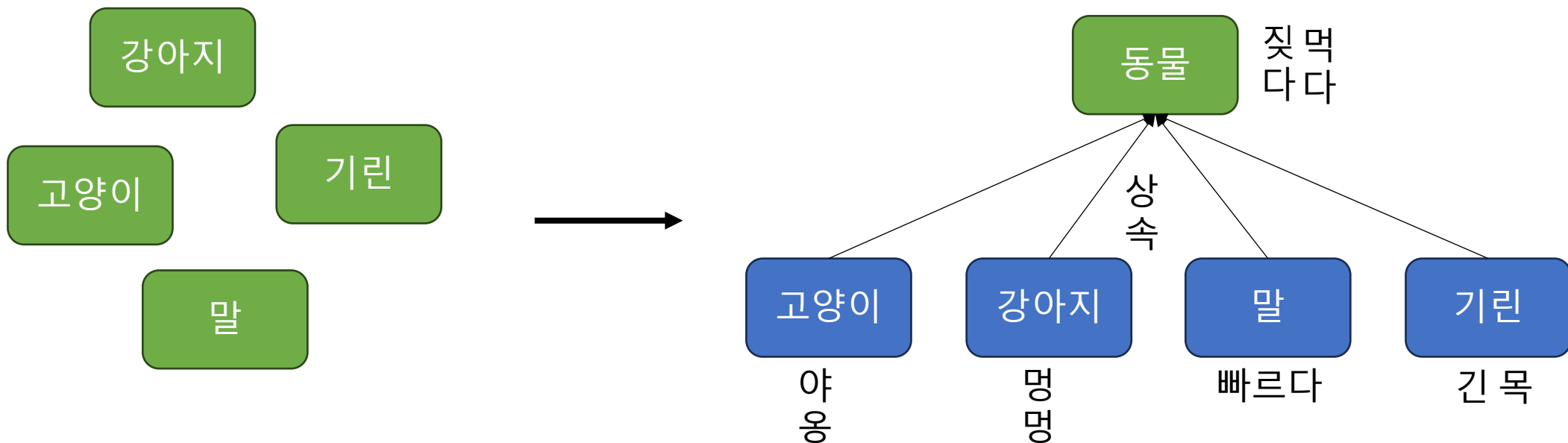


**상속**

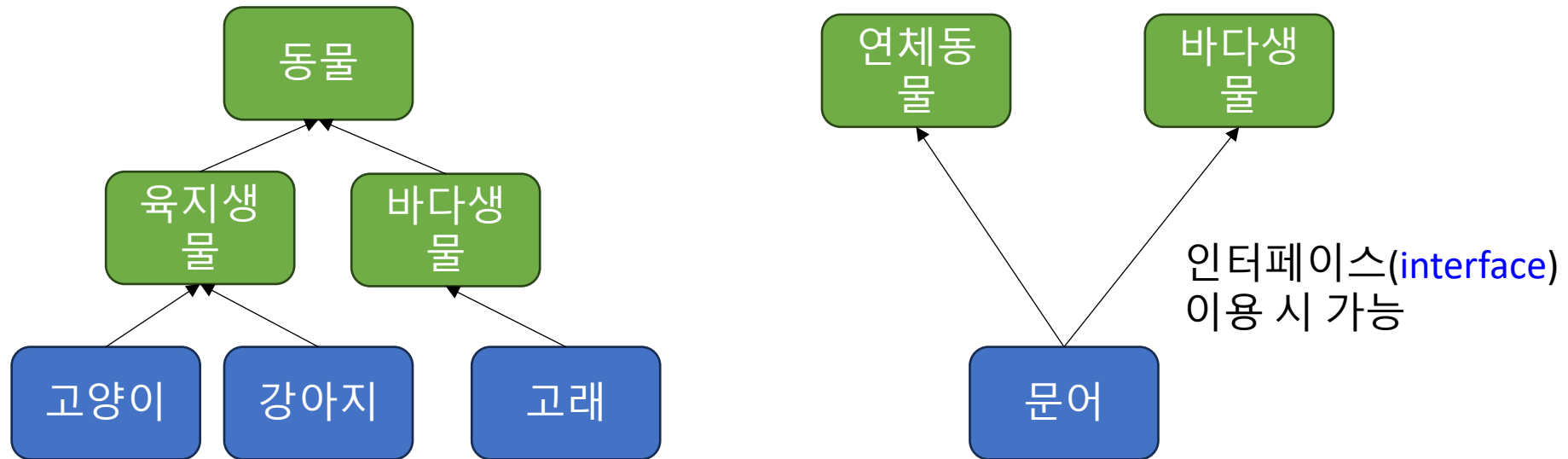
# 클래스 상속

- 하나의 클래스(부모 클래스)의 속성과 메서드를 다른 클래스(자식 클래스)가 물려받아 사용
- 객체 간의 공통된 필드 및 메서드의 코드 중복을 줄이고 유지보수를 쉽게 하기 위해서



# 클래스 상속

- 여러 차례에 걸쳐 상속 하는 것은 가능하나, 한 번에 여러 개의 상속을 받는 것은 불가능 (대신 인터페이스를 이용)
- 클래스 설계의 핵심은 **상속 관계를 어떻게 나눌지** 결정하는 것이라고도 할 수 있음



# 클래스 상속 구조

- (자식 클래스 이름) : (상속 받을 부모 클래스 이름)
- 부모 클래스의 public, internal, protected로 선언된 부분을 사용 가능

```
// 부모 클래스
3 references
public class Shape
{
    1 reference
    protected string getShape() => "Shape";
    0 references
    public string getShape2() => "Shape2";
    0 references
    private string getShape3() => "Shape3";
    0 references
    internal string getShape4() => "Shape4";

    protected int a;
    public int b;

    1 reference
    public int c { get; set; }
    private int d;
}
```

```
// 자식 클래스
3 references
public class Square : Shape
{
    // getShape(): 부모 클래스의 것
    1 reference
    public Square() => MessageBox.Show(getShape());

    0 references
    void test()
    {
        // 부모 클래스의 변수
        a = 1;
        b = 2;
        c = 3;
    }
}
```

# 클래스 상속

- 여러 차례 클래스를 상속 받는 것도 가능

```
2 references
public class GrandParent
{
    0 references
    public GrandParent() { }
}
```

```
2 references
public class Parent : GrandParent
{
    0 references
    public Parent() { }
}
```

```
4 references
public class Child : Parent
{
    1 reference
    public Child() { }
}
```

```
1 reference
public class GrandChild : Child
{
    0 references
    public GrandChild() { }
}
```

# 클래스 상속

- 부모에게 생성자가 있을 경우 부모의 생성자가 먼저 실행

```
1 reference
void Test()
{
    Child child = new Child();
}
```

```
2 references
public class Parent
{
    // 부모 생성자가 먼저 호출됨
    0 references
    public Parent() { }
}
```

```
3 references
public class Child : Parent
{
    // 자식 생성자가 나중에 호출됨
    1 reference
    public Child() { }
}
```

# 클래스 상속

- 자식이 부모 생성자를 실행하고 싶을 경우 base 키워드 사용
- base 키워드를 사용해도 부모 생성자가 먼저 호출됨

```
6 references
public class Parent
{
    // 부모 클래스의 생성자 -> 자식 클래스의 생성자 순서로 실행
    0 references
    public Parent() { }
    1 reference
    public Parent(string name) { }
    1 reference
    public Parent(string name, int id) { }
}
```

```
4 references
public class Child : Parent
{
    // base를 사용하여 부모 생성자를 호출
    // Parent(string name) 실행 후 Child() 실행
    1 reference
    public Child() : base("John") { }

    // 자식 생성자가 입력받은 값을 그대로 부모 생성자에 전달
    0 references
    public Child(string name, int id) : base(name, id) { }
}
```

# 클래스 상속

- 부모에게 기본 생성자(입력 값이 없는 생성자)가 없고, 입력 값이 있는 생성자만 있을 경우 자식 클래스에서 반드시 base 키워드를 사용하여 부모 클래스의 생성자를 호출해야 함

```
3 references
public class Parent
{
    0 references
    public Parent(string name) { }
    0 references
    public Parent(string name, int id) { }
}
```

```
3 references
public class Child : Parent
{
    // 부모에게 기본 생성자가 없는데 자식에서
    // base를 사용하지 않으면 오류 발생
    1 reference
    public Child() { }
}
```



# 업캐스팅 & 다형성

# 업캐스팅 (UpCasting)

- 자식 클래스의 객체를 부모 타입으로 참조하는 것.
- 반대의 경우(다운캐스팅)도 특정 조건을 만족하면 가능하지만 일반적으로 사용되지는 않음
- 보통 하나의 타입(상위 개념) 객체로 여러 종류의(하위 개념) 객체를 바꿔가면서 사용할 때 활용됨
- 예) 화폐(부모), 원(자식), 달러(자식), 위안(자식)
  - 화폐 = new 원
  - 화폐 = new 달러
  - 화폐 = new 위안

```
Money m = new Won();      // 업캐스팅  
Money m = new Dollar();   // 업캐스팅  
Money m = new Yuan();     // 업캐스팅
```

# 업캐스팅 (UpCasting)

6 references

```
public class Weapon
{
    protected int damage;

    2 references
    public Weapon(int damage)
    { this.damage = damage; }

    2 references
    public int Attack() => damage;
}
```

3 references

```
public class Sword : Weapon
{
    int attack_range = 1;

    // base()를 통해 부모 생성자에게 전달됨
    1 reference
    public Sword(int damage) : base(damage) {}

    1 reference
    public int Slash(int range)
    {
        if (this.attack_range >= range)
        {
            return this.damage;
        }
        return 0;
    }
}
```

3 references

```
public class Gun : Weapon
{
    int attack_range = 10;

    1 reference
    public Gun(int damage) : base(damage) { }

    1 reference
    public int Fire(int range)
    {
        if(this.attack_range >= range)
        {
            return this.damage;
        }
        return 0;
    }
}
```

# 업캐스팅 (UpCasting)

// 자식 -> 부모, 업캐스팅

```
Weapon weapon = new Sword(200);
```

// 부모의 메소드는 평범하게 이용 가능

```
int attack = weapon.Attack();
```

// 자식의 메소드는 자식 클래스로 캐스팅하여 사용

```
int skill_attack = ((Sword)weapon).Slash(1);
```

// 인스턴스를 덮어 쓰는 것으로 자식 클래스를 변경 가능

```
weapon = new Gun(100);
```

```
attack = weapon.Attack();
```

```
skill_attack = (((Gun)weapon).Fire(10));
```

# 실습. 클래스 업캐스팅

- Animal 클래스(부모)
- Dog, Cat, Bird 클래스(자식) : 모두 Animal을 상속
- Animal 클래스에 이름(Name)과 Speak() 메서드 선언.
  - Ex) (Name): 동물이 소리를 냅니다.
- 각 자식 클래스는 Name 속성을 설정하고, **자식 고유의 메서드**를 각각 하나씩 추가.
- Main에서 자식 객체를 생성한 후, 업캐스팅하여 Animal 타입 변수로 참조.
- 업캐스팅된 객체로 호출 가능한 메서드를 확인할 것! (주석으로 표시)
- 호출 **불**가능한 메서드는 다운캐스팅하여 호출 할 것! (주석으로 표시)
- **GitHub Repo. URL**과 **콘솔 결과창**을 슬랙 댓글로 제출

# 오버로드 & 오버라이드

# 메소드 오버로드

- 같은 이름의 메서드를 매개변수만 다르게 여러 개 정의 하는 것

2 references

```
public class MethodOverload
{
    1 reference
    public int GetNumber(int number)
    {
        return number;
    }

    1 reference
    public int GetNumber(int num1, int num2)
    {
        return num1 + num2;
    }

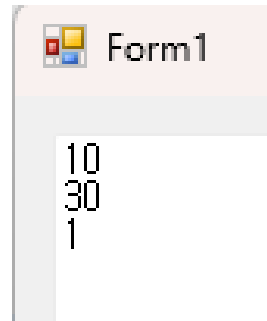
    1 reference
    public long GetNumber(long number)
    {
        return number - 99;
    }
}
```

1 reference

```
public Form1()
{
    InitializeComponent();

    MethodOverload mo = new MethodOverload();

    textBox_print.Text += mo.GetNumber(10).ToString() + "\r\n";
    textBox_print.Text += mo.GetNumber(10, 20).ToString() + "\r\n";
    textBox_print.Text += mo.GetNumber((long)100).ToString() + "\r\n";
}
```



Form1

10  
30  
1

# 메소드 오버라이드

- 상속 관계에서 부모 클래스의 메소드와 같은 이름의 메소드를 자식 클래스에서 작성하여 부모 메소드의 기능 대신 작동시킴

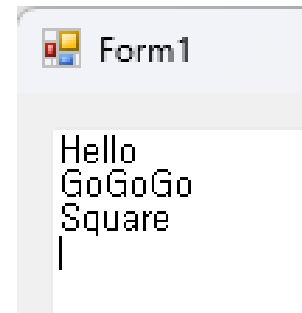
```
1 reference
public class Shape
{
    0 references
    public string Say() => "Hi\r\n";
    0 references
    public string Go() => "Go\r\n";
    2 references
    public virtual string getShape() => "Shape\r\n";
}

2 references
public class Square : Shape
{
    1 reference
    public string Say() => "Hello\r\n";
    1 reference
    public new string Go() => "GoGoGo\r\n";
    2 references
    public override string getShape() => "Square\r\n";
}
```

```
1 reference
public Form1()
{
    InitializeComponent();

    Square square = new Square();

    textBox_print.Text += square.Say();
    textBox_print.Text += square.Go();
    textBox_print.Text += square.getShape();
}
```



Form1

Hello  
GoGoGo  
Square  
|



# 가상 메소드

- virtual 키워드를 사용하여 업캐스팅을 할 경우 자식의 메소드가 실행되도록 함

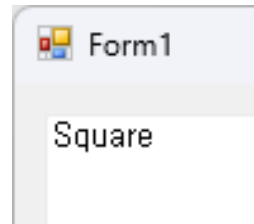
```
1 reference
public class Shape
{
    2 references
    public virtual string getShape() => "Shape\r\n";
}

2 references
public class Square : Shape
{
    2 references
    public override string getShape() => "Square\r\n";
}
```

```
1 reference
public Form1()
{
    InitializeComponent();

    Shape square = new Square();

    textBox_print.Text += square.getShape();
}
```



# 오버라이드 방지

- 상속이 여러 차례 이루어질 때, sealed 키워드를 사용하여 자식 클래스가 오버라이드를 할 수 없도록 방지

```
2 references
public class Shape
{
    3 references
    public virtual string getShape() => "Shape\r\n";
}

2 references
public class Square : Shape
{
    2 references
    public override sealed string getShape() => "Square\r\n";
}
```

```
0 references
public class Ring : Square
{
    3 references
    public override string getShape() => "Ring\r\n";
}

0 references
public class MethodOverload
{
    0 references
    public int GetNumber(int num...)
}
```

string Ring.getShape()

CS0239: 'Ring.getShape()': cannot override inherited member 'Square.getShape()' because it is sealed

Show potential fixes (Alt+Enter or Ctrl+.)

# 다형성이란? (개념)

- 같은 타입(부모 타입)으로 여러 형태(자식 객체)를 다룰 수 있는 능력.

```
class Weapon
{
    public virtual void Attack()
    {
        Console.WriteLine("무기로 공격합니다.");
    }
}

class Sword : Weapon
{
    public override void Attack()
    {
        Console.WriteLine("검으로 베기!");
    }
}

class Gun : Weapon
{
    public override void Attack()
    {
        Console.WriteLine("총으로 발사!");
    }
}
```

```
Weapon w1 = new Sword(); // 업캐스팅
Weapon w2 = new Gun();    // 업캐스팅

w1.Attack(); // 검으로 베기!
w2.Attack(); // 총으로 발사!
```

# 다형성

- 조건
  - 1) 상속
  - 2) 업캐스팅
  - 3) 오버라이드
- Why?
- 코드 유지보수의 핵심

# 인터페이스

# 인터페이스

- "클래스들 간의 약속"을 정의하여, 동일한 메서드 구조를 갖게 하는 도구
- 메소드, 이벤트, 프로퍼티만 가질 수 있음 (생성자도 사용할 수 없음)
- 인터페이스를 상속 받는 클래스들은 반드시 인터페이스의 메소드들을 **오버라이딩** 해야함
- 모든 요소가 public으로 작동
- 메소드의 출력, 식별자, 이름은 있지만 **기능 구현을 안함** (오버라이딩을 강제하는 이유)
- 클래스와 다르게 여러 개의 인터페이스를 상속 받는 것이 가능

# 인터페이스

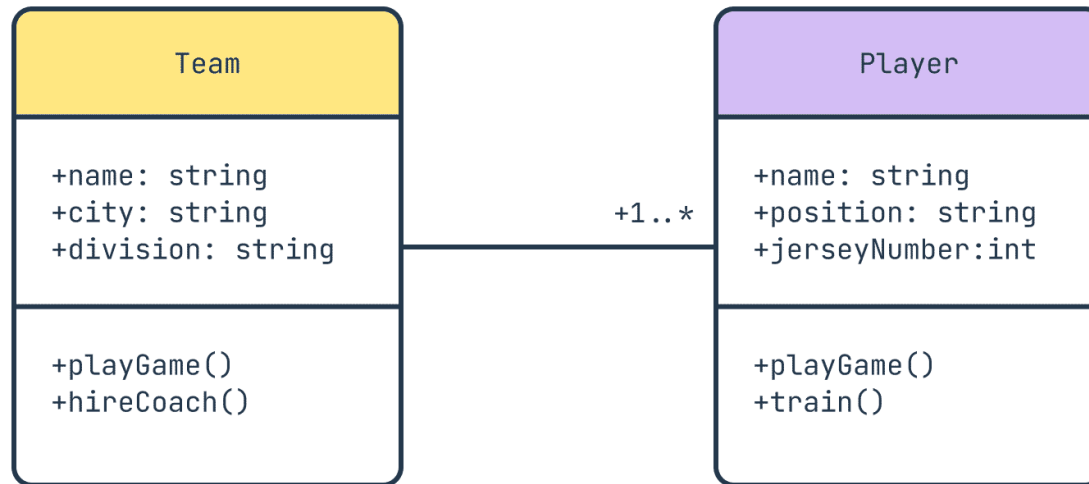
```
1 reference
public interface IHuman
{
    // 인터페이스는 기능 구현을 하지 않음
    1 reference
    void Talk();
}
```

```
1 reference
public interface IParent
{
    1 reference
    void GetChild();
}
```

```
// 인터페이스는 다중 상속이 가능
2 references
public class Child : IHuman, IParent
{
    // 인터페이스의 메소드를 반드시 오버라이딩 해야함
    1 reference
    public void Talk() { }
    1 reference
    public void GetChild() { }
}
```

# 클래스 다이어그램 (UML)

- <https://app.diagrams.net/>
- 클래스 설계를 문서로 그릴 때 사용하는 표준 표기 방법
- UML 작성 도구에 따라 자동으로 소스코드를 생성해주는 기능을 포함





# 종합 실습. 클래스 상속 설계

- 간단한 롤플레잉 게임을 설계하고 **UML**로 그려봅시다
- 게임에는 플레이어가 직접 조작하는 **주인공 캐릭터**, 플레이어와 상호작용을 하는 **NPC**(직접 조작 할 수 없는 캐릭터), 플레이어와 전투를 하는 **몬스터**가 존재
- 플레이어는 **전사**, **마법사** 두 가지 직업 선택이 가능
- 몬스터는 **오크**, **슬라임** 두 종류가 존재
- 전투는 **체력**, **공격력**이 존재하고, **공격력에 따라 상대방의 체력을 깎을 수 있음**
- draw.io의 UML을 png로 내보내기 하고, 이미지 파일을 슬랙 댓글로 제출

# 종합 실습. 클래스 상속

- 앞서 만들었던 롤플레이팅 게임 UML을 참고하여 소스코드 작성
- 기능 구현이 불가능한 부분(플레이어 조작 등)은 메소드 이름만 작성하고 기능 구현은 하지 않아도 됨!
- 업캐스팅을 **최소 1회 이상 사용**
- 전투는 플레이어-몬스터, NPC-몬스터 간에 가능 (플레이어-NPC는 불가능)
- 플레이어와 몬스터가 전투하는 부분을 코드로 작성
- GitHub Repo. URL을 슬랙 댓글로 제출

# 종합 실습. 오버라이딩, 오버로딩

- 앞서 만든 RPG 게임에서 부모 클래스에 Talk() 메소드를 선언하고 자식 클래스는 오버라이딩 함 (**virtual, override 사용**)
  - 캐릭터 별로 어울리는 대사를 MessageBox로 출력
- 플레이어에게 LevelUp() 메소드를 만들고 오버로딩
  - hp만 입력 받으면 hp 수치만 변경
  - hp 및 power를 입력 받으면 hp 및 power 수치를 변경
  - 아무것도 입력 받지 않았다면 적당한 오류 메시지를 반환
- 위 두개 코드를 모두 테스트
- GitHub Repo. URL을 슬랙 댓글로 제출