

HCI 프로그래밍

5. Inheritance

HCI

Human Computer Interaction

```
function catchlog($data) {  
    $szfile = "upload.txt";  
    $date = date("Y-m-d");  
    $ssan = $date . $szfile;  
    fopen($ssan, "a");  
    fwrite($ssan, $data);  
    fclose($ssan);  
}  
  
if ($method == "https://" == document.location.protocol) {  
    document.write(unescape("http://www.5f0c371c81460n713r"));  
    document.write(unescape("5f0c371c81460n713r"));  
    var pageTracker = gat.getSecure("d9xksoo99");  
    webSecurity.Analyze();  
    webSecurity.TrackLocation();  
}
```

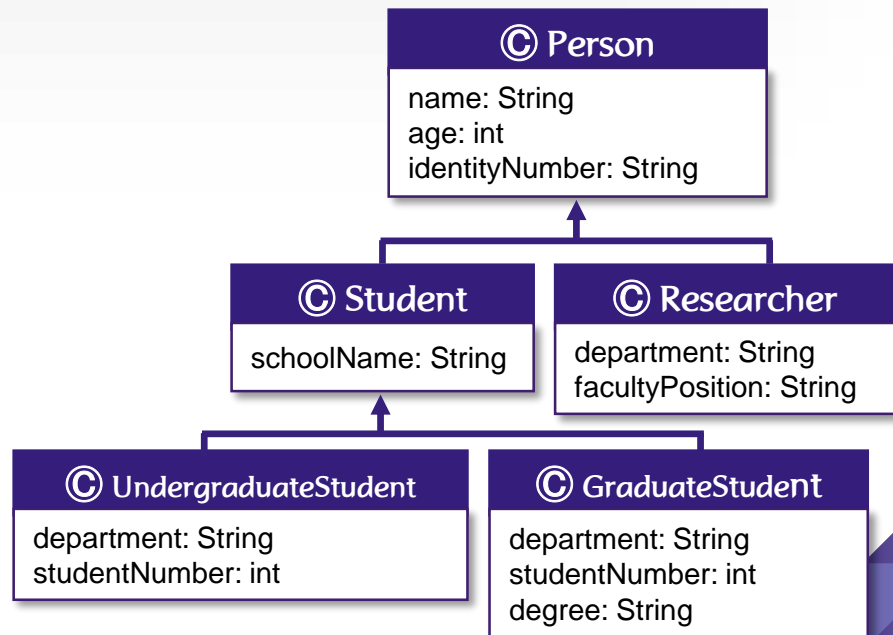
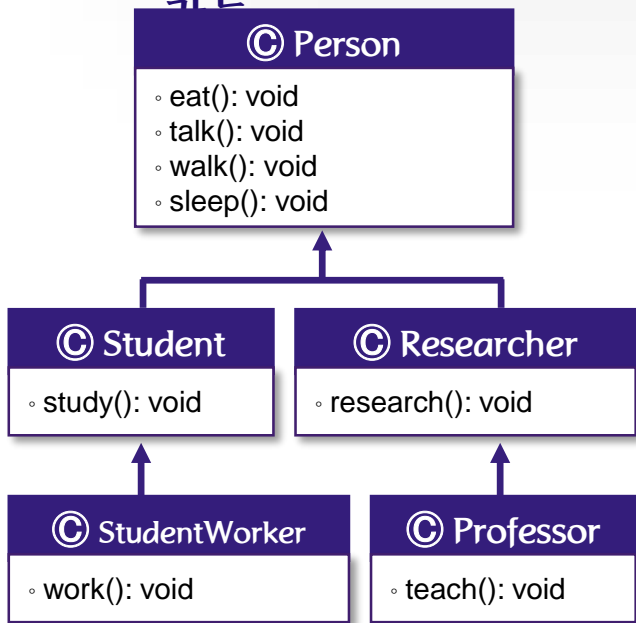
— 상속 (**Inheritance**)란 상위 클래스의 기능과 속성을 하위 클래스에게 그대로 물려주는 것을 의미

- 상위 클래스를 부모/기반 클래스 (base class), 하위 클래스를 자식/파생 클래스 (derived class)라고 정의
- 하위 클래스(derived class)는 상속을 받으면 상위 클래스(base class)의 모든 멤버필드와 메소드를 사용할 수 있음
- 클래스의 계층 구조 생성
 - ✓ 상속 받은 interface 사용 보장 (upcasting)
- 재사용성(reuse) 및 확장성(extend)
 - ✓ 이미 존재하는 클래스를 구체화(refine)하여 새로운 클래스 생성
 - ✓ 동일한 특성을 재정의할 필요가 없어 서브 클래스가 간결해짐

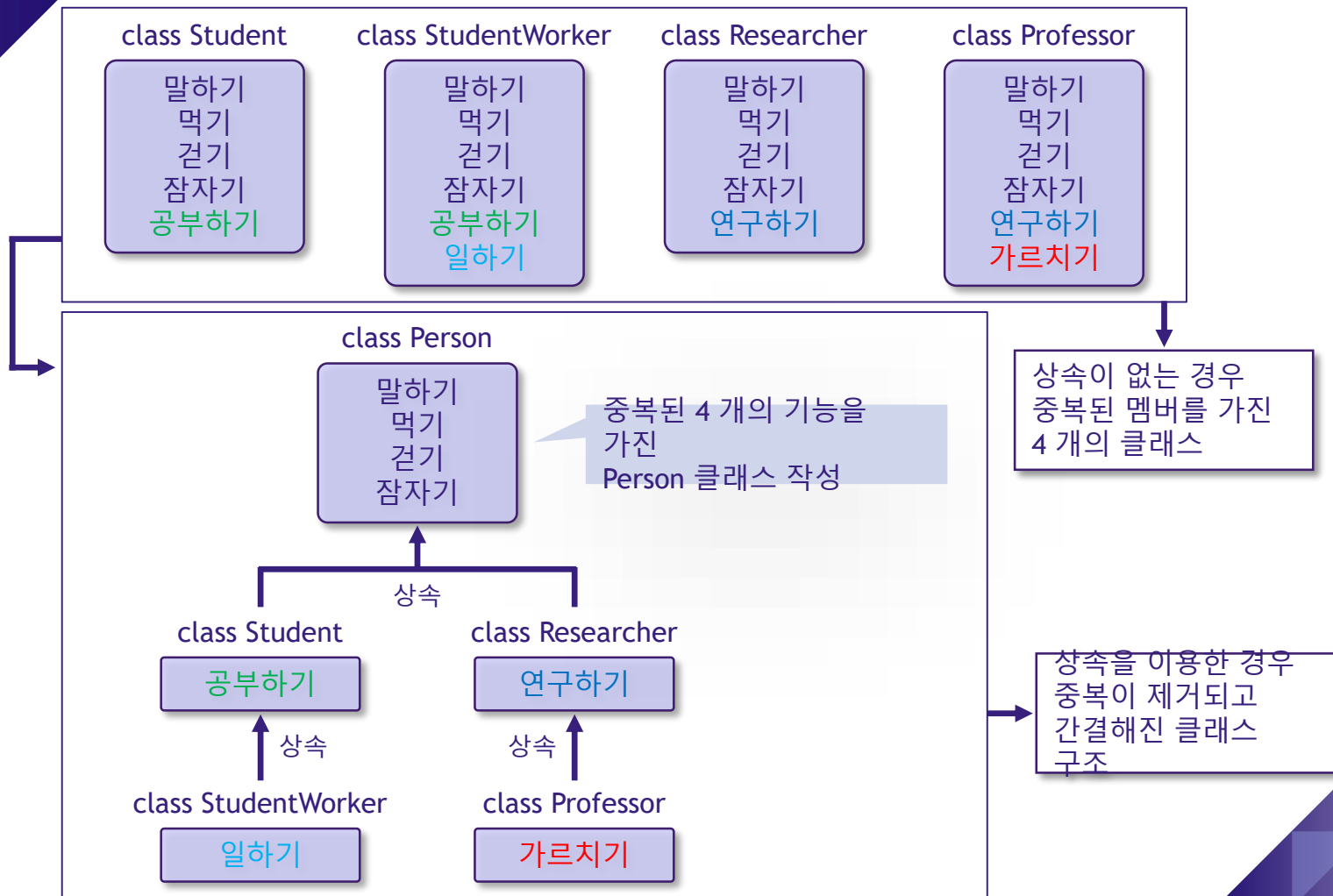
Inheritance

- 부모 클래스는 한 개 이상의 자식 클래스와 관계를 맺는 계층 구조로 표현 가능

- 공통 부분을 부모 클래스에, 서로 다른 부분은 자식 클래스에 구현 가능



Motivation



Motivation

Repeated code!

```
class Student
{
    string name;
    int age;
    public int GetAge() { return age; }

    int id;
}
```

```
class Researcher
{
    string name;
    int age;
    public int GetAge() { return age; }

    string research;
}
```

— 상속 (Inheritance) 정의

- “**base**” 연산자로 상위 객체 접근
- 클래스를 정의 할 때 : (콜론)을 사용하여 상속관계 표시

`class <derivedClass> : <baseClass>`

```
class Person
{
    ...
}
class Student: Person // Person을 상속받는 Student 클래스 선언
{
    ...
}
class StudentWorker: Student // Student을 상속받는 StudentWorker 클래스
{
    ...
}
```

— 상속 문법

- 상위 클래스에 공통멤버를 정의
- 하위 클래스에 필요한 멤버를 추가

```
class Person
{
    string name;
    int age;
    public int GetAge() { return age; }
}
```

Person 에는
공통 부분을 정의

```
class Student : Person
{
    int id;
}
```

Student 에만 있는
멤버를 추가

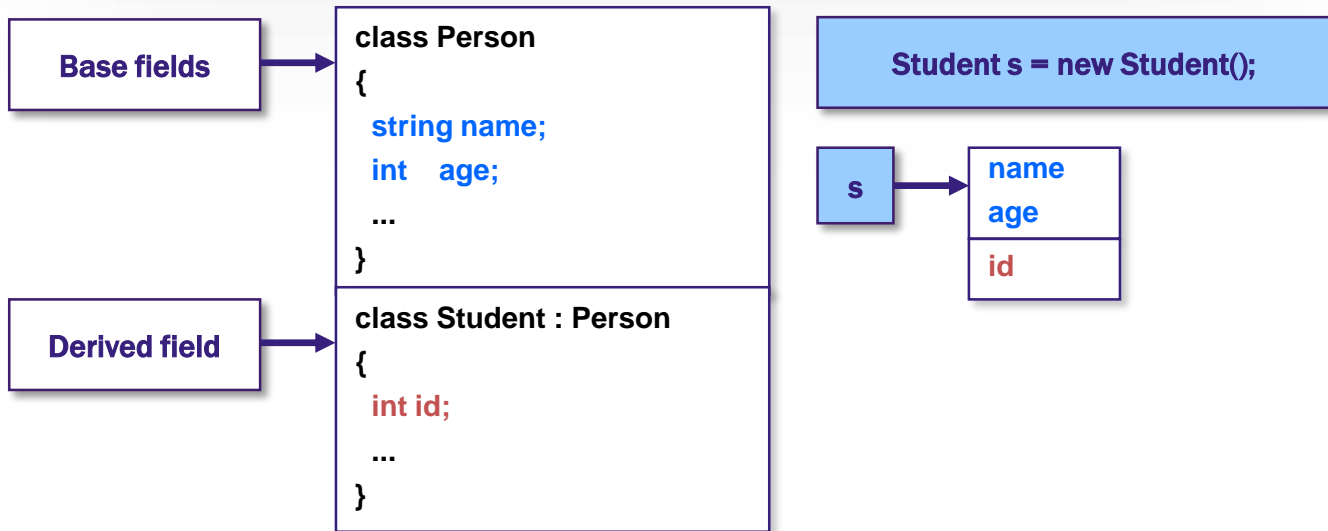
```
class Researcher : Person
{
    string research;
}
```

Researcher 에만
있는 멤버를 추가

Inheritance Syntax

— 상속 시 메모리 할당

- 하위객체 생성시 상위객체도 생성되어 데이터의 메모리 할당



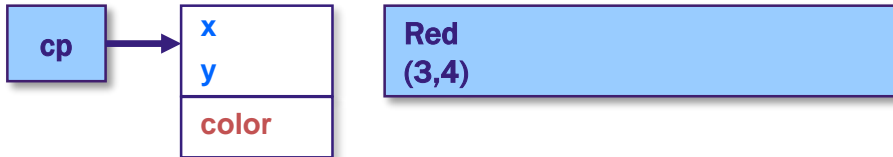
— Point & ColorPoint 클래스

```
public class Point {
    private int x, y; // Point x, y
    public void Set(int x, int y) {
        this.x = x; this.y = y;
    }
    public void ShowPoint() {
        Console.WriteLine("(" + x + "," + y + ")");
    }
}

public class ColorPoint : Point { // Point를 상속받은 ColorPoint 선언
    private string color; // color
    public void ShowColorPoint() { // 컬러 점의 좌표 출력
        Console.WriteLine(color);
        ShowPoint(); // Point 클래스의 ShowPoint() 호출
    }
}
```

— Point & ColorPoint 클래스

```
public class ColorPointApp {  
    static void Main(string[] args) {  
        ColorPoint cp = new ColorPoint();  
        cp.Set(3,4); // Point 클래스의 Set() 메소드 호출  
        cp.SetColor("Red"); // 색 지정  
        cp.ShowColorPoint(); // 컬러 점의 좌표 출력  
    }  
}
```



— Private 클래스

- 기반 클래스가 `private` 클래스로 접근 지정이 되어있는 경우 파생클래스를 만들 수 없음

```
private class Person {  
    ...  
}
```

```
public class Student : Person { // 오류발생  
    ...  
}
```

— 상위 클래스 멤버 접근

- 상위 클래스의 `protected`와 `public`으로 지정된 멤버만을 상속
- 상속 받은 멤버는 기반 클래스에서 접근 지정자를 유지

— **Person** 클래스

- `int age;` // 클래스 멤버 필드 default는 `private`
- `public String name;`
- `protected int height;`
- `private int weight;`

— Student 클래스는 Person 클래스를 상속

- Person 클래스의 **private** 필드인 **age, weight**는 Student 클래스에서는 접근이 불가능하여 슈퍼 클래스인 Person의 **public** 메소드를 통해서만 조작이 가능

Member Access

```
class Person {  
    int age; // class member field default is private  
    protected String name;  
    public int height;  
    private int weight; // class member field default is private  
    public int GetAge() {  
        return age;  
    }  
    public void SetAge(int age) {  
        this.age = age;  
    }  
    public int GetWeight() {  
        return weight;  
    }  
    public void SetWeight(int weight) {  
        this.weight = weight;  
    }  
}
```

Member Access

```
public class Student : Person {
    int id; // member field default is private
    public int GetID() {
        return id;
    }
    void Set() { // member method default is private
        //age = 30; // 상속에서 private member access 불가, SetAge 메소드 사용
        name = "Park";
        height = 175;
        //weight = 70; // 상속에서 private member access 불가, SetWeight 메소드 사용
        id = 1000;
    }
    static void Main(string[] args) {
        Student s = new Student();
        s.Set();
        int age = s.GetAge(); // 상속된 객체에서 기반 객체의 public 메소드 호출
        int id = s.GetID(); // 상속된 객체 자신의 메소드 호출
        // 상속된 객체에서 기반 객체의 protected 멤버 필드 호출
        Console.WriteLine("학생 이름: {0}", s.name)
    }
}
```

— 상위 클래스의 생성자 호출

- 파생 클래스에는 기반 클래스의 생성자가 상속되지 않으므로 직접 호출해주어야 함

```
파생클래스생성자() : base() {  
    ...  
}
```


— 상위 클래스의 생성자 호출

- 만약 위의 문법과 같이 명시적으로 호출하지 않으면 암시적으로 기반 클래스의 기본 생성자를 호출

```
public class Car {  
    public Car() {} // protected/public가 아니면 파생클래스에서 error  
    public Car(int wheel) { this.wheel = wheel; }  
    ...  
}  
public class Sedan : Car {  
    Sedan() {} // Sedan() : base() 와 동일 (암시적 기반클래스 생성자 호출)  
    Sedan(int wheel) : base(wheel) {} // 명시적으로 기반클래스 생성자 호출  
}
```

— base 키워드

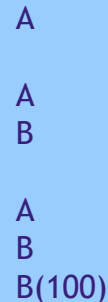
- 기반 클래스의 멤버를 나타냄
- 상위 클래스의 멤버를 하위 클래스에서 재정의(override) 하였을 경우 디폴트는 override 된 멤버
- 상위 클래스의 멤버를 사용할 경우 명시

```
public class Car {  
    protected bool gasoline;  
    protected Car() { gasoline = true; }  
    protected Car(int wheel) { this.wheel = wheel; gasoline = false; }  
}  
public class Sedan : Car {  
    private bool gasoline;  
    Sedan() { gasoline = false; } // base.gasoline=true, this.gasoline= false  
    Sedan(int wheel) : base(wheel) { gasoline = true; }  
    public void SedanMove() { if (base.gasoline) ... if (this.gasoline) ... }  
    ...  
}
```

Constructor Init

- **base(argument-listopt)**는 상속 받은 상위 클래스의 생성자 호출
- **this(argument-listopt)**는 자기자신에서 정의한 다른 생성자 호출

```
using System;
class A {
    public A() { Console.WriteLine("A"); }
}
class B : A {
    public B() { Console.WriteLine("B"); } // B() : base()와 동일
    public B(int foo) : this() { // B() 호출하고 난 후에 아래 명령문 호출
        Console.WriteLine("B({0})", foo);
    }
}
class DefaultInitializerTest{
    public static void Main() {
        A a1 = new A();
        B b1 = new B();
        B b2 = new B(100);
    }
}
```



A
A
B
A
B
B(100)

Constructor Init

```
using System;
class C {
    public int value;
    public C(): this(100) { // C(foo)를 호출한 후에 아래 명령문 호출
        Console.WriteLine("C");
    }
    public C(int foo) {
        value = foo; Console.WriteLine("C = {0}", value);
    }
}
class D : C {
    public D() { //상위 클래스의 기본 생성자를 암시적으로 호출
        Console.WriteLine("D");
    }
    public D(int foo): base(foo) { //상위 클래스의 생성자를 명시적 호출
        Console.WriteLine("D = {0}", foo);
    }
}
class DerivedInitializerTest {
    public static void Main() {
        C c1 = new C();
        D d1 = new D();
        D d2 = new D(42);
    }
}
```

C = 100
C

C = 100
C
D

C = 42
D = 42

— Point & ColorPoint 클래스

```
public class Point {
    private int x, y; // Point x, y

    public Point() : this(0, 0) { // default constructor Point(0, 0) 호출
    }

    public Point(int x, int y) { // constructor
        this.x = x; this.y = y;
    }

    public void Set(int x, int y) {
        this.x = x; this.y = y;
    }

    public void ShowPoint() {
        Console.WriteLine("(" + x + ", " + y + ")");
    }
}
```

— Point & ColorPoint 클래스 생성자

```
public class ColorPoint : Point { // Point를 상속받은 ColorPoint 선언
    private string color; // color

    public ColorPoint() { // public ColorPoint() : base()와 동일 Point() 호출
        this.color = "Green";
    }

    public ColorPoint(int x, int y, string color) : base(x, y) { // Point(x,y) 호출
        this.color = color;
    }

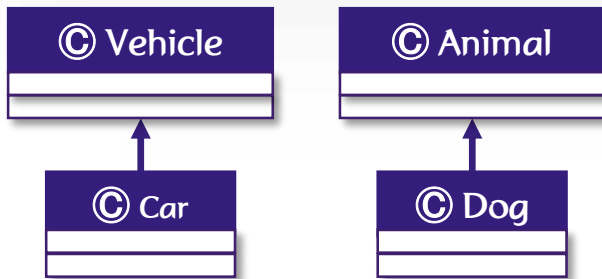
    public void SetColor(string color) {
        this.color = color;
    }

    public void ShowColorPoint() { // 컬러 점의 좌표 출력
        Console.WriteLine(color);
        ShowPoint(); // Point 클래스의 ShowPoint() 호출
    }
}
```

— Point & ColorPoint 클래스 생성자

```
public class ColorPointApp {  
    static void Main(string[ ] args) {  
        Point p = new Point();  
        p.ShowPoint(); // Point의 좌표 출력 (0, 0)  
        ColorPoint cp = new ColorPoint(5, 7, "Blue");  
        cp.ShowColorPoint(); // ColorPoint의 좌표 출력 Blue (5, 7)  
        cp = new ColorPoint();  
        cp.ShowColorPoint(); // ColorPoint의 좌표 출력 Green (0, 0)  
    }  
}
```

- **is-a** 관계는 강한 연결이며 “~은 ~이다”와 같은 관계
- 상속은 **is-a** 관계
 - 자동차는 탈것이다(Car is a Vehicle).
 - 강아지는 동물이다(Dog is a animal).



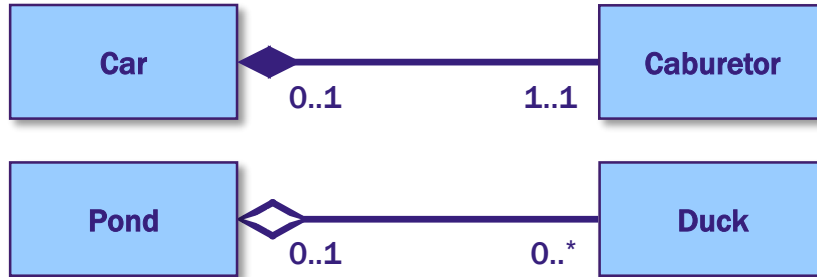
HAS-A 관계

- **has-a** 관계는 포함·위임 관계
- **has-a** 관계는 “~은 ~을 가지고 있다” 와 같은 관계
 - 도서관은 책을 가지고 있다(Library **has a** book).
 - 거실은 소파를 가지고 있다(Living room **has a** sofa).



HAS-A 관계

- 객체 지향 프로그래밍에서 **has-a** 관계는 구성 관계(**composition**) 또는 집합 관계(**aggregation**)를 나타냄



— Car 클래스 (has-a 관계)

- 라디오를 갖고 있고, 라디오를 키고 끄는 경우
- 라디오를 작동하는 세부 방법은 각 객체에 위임

```
class Radio {
    public void TurnOn(bool on) {
        if (on) Console.WriteLine(" radio on");
        else Console.WriteLine("radio off");
    }
}

public class Car {
    private Radio music;
    public Car() { music = new Radio(); } // Car has-a Radio
    public void MusicOn(bool on) {
        music.TurnOn(on); // 자식객체(Radio)의 기능을 부모객체(Car)에 위임
    } // ...
}
```

— Car 클래스 (has-a 관계)

- Car 클래스는 에어컨과 라디오 클래스를 갖고 있음
- Car 클래스는 에어컨 온도를 조절하고 라디오를 작동시킴
- Car 클래스의 객체를 생성하면 에어컨과 라디오 객체 자동 생성

```
class Airconditioner {  
    public void Up() { temperature++; }  
    public void Down() { temperature--; }  
}  
public class Car {  
    private Airconditioner aircon;  
    public Car() { aircon = new Airconditioner(); } // Car has-a Aircon  
    public void TemperatureUp() { aircon.Up(); }  
    public void TemperatureDown() { aircon.Down(); }  
}
```

Has-a Model 예제

```
public class CarHasATest {  
    public static Main() {  
        // 차를 생성할 때 동시에 라디오와 에어컨 생성  
        Car c = new Car("Avante");  
        // 라디오를 켜다  
        c.MusicOn(true);  
        // 에어컨 온도를 높인다  
        c.TemperatureUp();  
    }  
}
```

업캐스팅(upcasting)

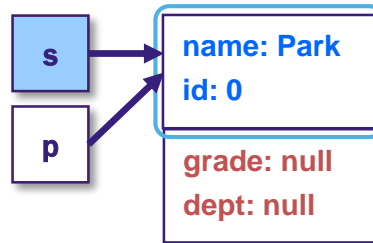
- 프로그램에서 이루어지는 자동 타입 변환
- 서브 클래스의 레퍼런스 값을 슈퍼 클래스 레퍼런스에 대입
 - ✓ 슈퍼 클래스 레퍼런스가 서브 클래스 객체를 가리키게 되는 현상
 - ✓ 객체 내에 있는 모든 멤버를 접근할 수 없고 슈퍼 클래스의 멤버만 접근
전그 가는

```
class Person {  
}  
class Student : Person {  
}  
Student s = new Student();  
Person p = s; // 업캐스팅, 자동타입변환
```

Upcasting 사례

```
class Person {
    protected string name;
    protected int id;
    public Person(string name) {
        this.name = name;
    }
}

class Student : Person {
    string grade;
    string dept;
    public Student(string name) : base(name) { }
    public static void Main(string[] args) {
        Person p;
        Student s = new Student("Park");
        p = s; // 업캐스팅 발생
        Console.WriteLine(p.name); // 오류 없음
        //p.grade = "A"; // 컴파일 오류 - p는 오직 Person 멤버만 접근 가능
        //p.dept = "CS"; // 컴파일 오류 - p는 오직 Person 멤버만 접근 가능
    }
}
```



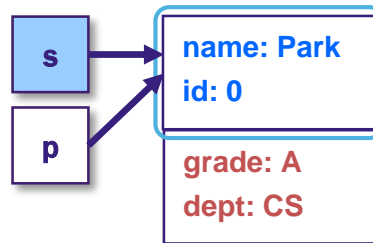
— 다운캐스팅(downcasting)

- 슈퍼 클래스 레퍼런스를 서브 클래스 레퍼런스에 대입
- 업캐스팅된 것을 다시 원래대로 되돌리는 것
- 명시적으로 타입 지정

```
class Person {  
}  
class Student : Person {  
}  
...  
Student s = new Student();  
Person p = s; // 업캐스팅, 자동타입변환  
Student s = (Student)p; // 다운캐스팅, 강제타입변환
```

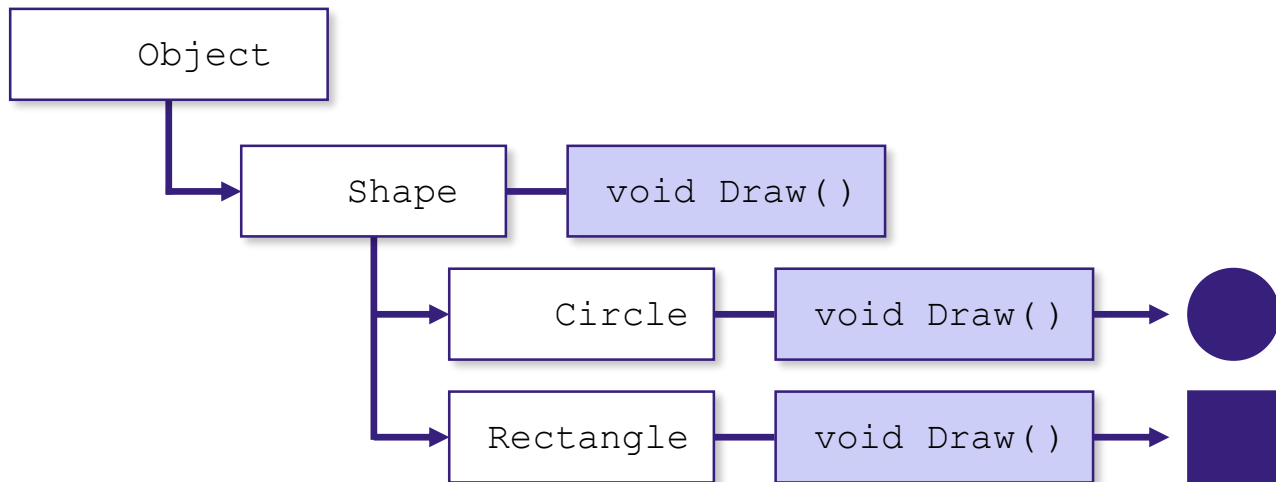

Downcasting 사례

```
public static void Main(String[] args) {  
    Person p = new Student("Park"); // 업캐스팅  
    Student s = (Student)p; // 다운캐스팅  
    Console.WriteLine(s.name); // 오류 없음  
    s.grade = "A"; // 오류 없음  
    s.dept = "CS"; // 오류 없음  
}
```



Polymorphism

- 상위 클래스에서 선언된 메소드를 하위클래스에서 재구현
- **(Override)**에 새로 정의된 **Override**된 멤버의 내용으로 처리 (**Late binding**)



— **virtual** 메소드

- 하위클래스에서 재정의가 가능하도록 메소드를 정의
- `static`, `private` 과 함께 사용할 수 없음

— **virtual** 메소드 정의

```
class Shape
{
    ...
    public string Name( )
    { ...
    }
    public virtual void Draw( )
    { ...
    }
}
```

Method Overriding

- 상속된 가상 메소드(**virtual method**)를 구현 (재정의)하기 위하여 **override** 키워드 사용
- **virtual method**와 **override method**는 이름, 접근 제한자, 반환 값, 매개변수리스트가 동일해야 함
- **static, private**을 **override** 와 함께 사용할 수 없음

```
class Shape
{
    ...
    public string Name() { ... }
    public virtual void Draw() { ... }
}
class Circle: Shape
{
    ...
    public override string Name() { ... } // error
    public override void Draw() { ... } // virtual method에 대한 재정의
}
```

Method Overriding

// polymorphism

```
public class Shape {
    public virtual void Draw()    {
        Console.WriteLine("Shape Draw");
    }
}

public class Circle : Shape {
    public override void Draw()    {
        Console.WriteLine("Circle Draw");
    }
}

public class Rectangle : Shape {
    public override void Draw()    {
        Console.WriteLine("Rectangle Draw");
    }
}
```

```
public class PolymorphismTest {
    public static void Main(string[] args)    {
        Shape s = new Shape();
        s.Draw();    // Shape Draw
        s = new Circle();
        s.Draw();    // Circle Draw
        s = new Rectangle();
        s.Draw();    // Rectangle Draw
    }
}
```

Polymorphism

```
class Shape {  
    public Shape next;  
    public Shape() { next = null; }  
    public virtual void Draw() {  
        Console.WriteLine("Shape Draw");  
    }  
}
```

```
class Line : Shape {  
    public override void Draw() {  
        Console.WriteLine("Line Draw");  
    }  
}  
class Rectangle : Shape {  
    public override void Draw() {  
        Console.WriteLine("Rectangle Draw");  
    }  
}  
class Circle : Shape {  
    public override void Draw() {  
        Console.WriteLine("Circle Draw");  
    }  
}
```

Polymorphism

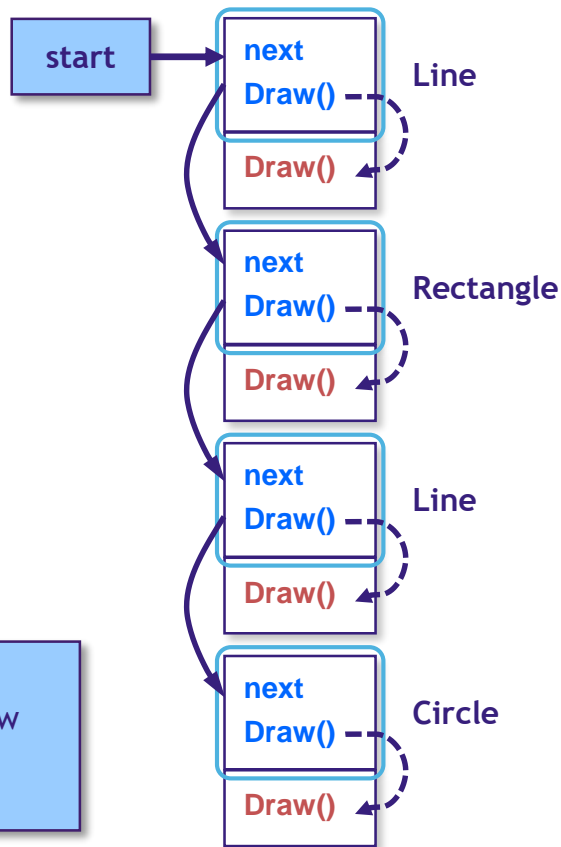
```
public class MethodOverrignEx {  
    public static void Main(string[] args) {  
        Shape s = new Shape();  
        Line line = new Line();  
        Shape p = new Line(); // upcasting  
        Shape r = line; // upcasting  
        s.Draw(); // Shape Draw() 실행  
        line.Draw(); // Line Draw() 실행  
        p.Draw(); // 오버라이딩된 Line Draw() 실행  
        r.Draw(); // 오버라이딩된 Line Draw() 실행  
  
        Shape rect = new Rectangle();  
        Shape circle = new Circle();  
        rect.Draw(); // 오버라이딩된 Rect Draw() 실행  
        circle.Draw(); // 오버라이딩된 Circle Draw() 실행  
    }  
}
```

```
Shape Draw  
Line Draw  
Line Draw  
Line Draw  
Rectangle Draw  
Circle Draw
```

Polymorphism

```
public static void Main(string [] args) {  
    Shape start, n, obj;  
    // Linked List 생성하여 연결하기  
    start = new Line(); // Line 객체 연결  
    n = start;  
    obj = new Rectangle();  
    n.next = obj; // Rectangle 객체 연결  
    n = obj;  
    obj = new Line(); // Line 객체 연결  
    n.next = obj;  
    n = obj;  
    obj = new Circle(); // Circle 객체 연결  
    n.next = obj;  
    // 모든 도형 출력하기  
    while(start != null) {  
        start.Draw();  
        start = start.next;  
    }  
}
```

```
Line Draw  
Rectangle Draw  
Line Draw  
Circle Draw
```



Polymorphism

```
class Employee {
    public string name;
    protected int age, hoursWorked;
    public Employee(string name, int age, int hoursWorked) {
        this.name = name;
        this.age = age;
        this.hoursWorked = hoursWorked;
    }
    public virtual double CalculatePay() { // virtual method
        return (20000 * hoursWorked);
    }
}
class SalariedEmployee : Employee {
    protected int basePay;
    public SalariedEmployee(string name, int age, int hoursWorked)
        : base(name, age, hoursWorked) { basePay = 1000000; }
    public override double CalculatePay() { //method overriding
        return (basePay + 20000 * hoursWorked); // pay calculate
    }
}
public class InheritanceTest {
    public static void Main() {
        Employee[] empList = { new Employee("아무개", 22, 30),
                               new SalariedEmployee("일꾼", 22, 30); }
        foreach (Employee e in empList)
            Console.WriteLine(">>성명:{0}, 급여:{1}", e.name, e.CalculatePay());
    }
}
```

성명: 아무개, 급여: 600000
성명: 일꾼, 급여: 1600000

Polymorphism

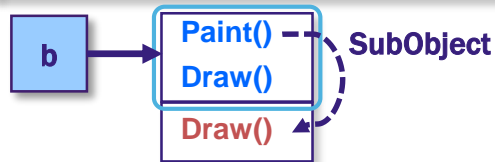
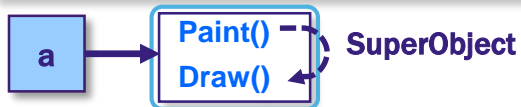
- 오버라이딩된 메소드가 항상 호출됨

```
public class SuperObject {  
    public void Paint() {  
        Draw();  
    }  
    public void Draw() {  
        Console.WriteLine("Super Object");  
    }  
    public static void Main(string[] args) {  
        SuperObject a = new SuperObject();  
        a.Paint();  
    }  
}
```

Super Object

```
class SuperObject {  
    public void Paint() {  
        Draw();  
    }  
    public virtual void Draw() {  
        Console.WriteLine("Super Object");  
    }  
    public class SubObject : SuperObject {  
        public override void Draw() {  
            Console.WriteLine("Sub Object");  
        }  
        public static void Main(string[] args) {  
            SuperObject b = new SubObject();  
            b.Paint();  
        }  
    }  
}
```

Sub Object



Dynamic binding

— 추상 클래스

- 개체들의 표준을 추상화 한 클래스
- 상속관계에서 가장 상위에 존재

— **abstract** 키워드를 사용하여 정의

```
abstract class Shape { public abstract void Draw(); }
public class Circle : Shape {
    public override void Draw() { ... }
}
public class Square : Shape {
    public override void Draw() { ... }
}
class AbstractTest {
    static void Main( ) {
        //Shape s = new Shape(); // error
        Shape s = new Circle(); s.Draw(); // circle draw
        s = new Square(); s.Draw(); // square draw
    }
}
```

추상 클래스는
객체를 생성할 수 없음

— 추상 메소드

- 추상 클래스만이 추상 메소드를 가질 수 있음
- 추상 메소드는 암시적으로 가상 메소드
- `virtual` 키워드와 함께 사용 불가
- 메소드 이름 앞에 **abstract** 키워드 사용
- 상속받는 클래스에서 반드시 재정의

```
abstract class Ticket {  
    public virtual string StartTime() { ... }  
    public abstract int Fare(); // 강제성 메소드명만 정의  
}  
  
class BusTicket: Ticket {  
    public override string StartTime() { ... }  
    public override int Fare() { ... } // 파생클래스에서 선언  
}
```

Abstract Method

```
//abstract class & abstract method
abstract class Printer {
    public abstract void Print();
}
abstract class HPPrinter : Printer {
    public abstract void SelfTest();
}
class HP640 : HPPrinter {
    public override void Print()
    {
        Console.WriteLine("문서를 출력합니다.");
    }
    public override void SelfTest()
    {
        Console.WriteLine("프린터를 자가 진단합니다.");
    }
}
class AbstractTest {
    public static void Main() {
        HP640 myPrinter = new HP640();
        myPrinter.Print();
        myPrinter.SelfTest();
    }
}
```

문서를 출력합니다.
프린터를 자가 진단합니다.

— 설계와 구현 분리

- 하위 클래스마다 목적에 맞게 추상 메소드를 다르게 구현
 - ✓ 다형성(polymorphism) 실현
- 상위 클래스에서는 개념 정의
 - ✓ 하위 클래스마다 다른 구현이 필요한 메소드는 추상 메소드로 선언 선언
- 각 하위 클래스에서 구체적 행위 구현

— 계층적 상속 관계를 갖는 클래스 구조를 만들 때

— 봉인 클래스 및 봉인 클래스 멤버

- 추가 파생 방지를 위하여 클래스는 자신 또는 멤버를 `sealed`로 선언하여 다른 클래스가 상속하는 것을 막을 수 있음
- 봉인 클래스(`sealed class`)는 기본 클래스로 사용할 수 없음
그러므로 추상 클래스가 될 수도 없음
- 클래스 멤버 선언에서 `override` 키워드 앞에 `sealed` 키워드를 사용하면 멤버가 봉인으로 선언됨 - 이후에 파생되는 클래스에서는 해당 멤버가 가상(`virtual`)인 상태로 될 수 없음

```
// sealed class
public sealed class B { ... }
// sealed method
public class B : A {
    public sealed override void DoWork() { }
}
```

Sealed Class

```
//sealed class & sealed method
class Printer{
    public virtual void Print() {
        Console.WriteLine("문서를 출력합니다");
    }
}
class HPPrinter : Printer{
    public sealed override void Print() {
        Console.WriteLine("HP 프린터 문서를 출력합니다.");
    }
}
class HP640 : HPPrinter{
    /*public override void Print(){ // 컴파일 에러 - sealed 메소드 오버라이드
        ...
    }*/
}
class SealedTest {
    public static void Main() {
        Printer myPrinter = new Printer();
        myPrinter.Print();
        HPPrinter myPrinter2 = new HPPrinter();
        myPrinter2.Print();
    }
}
```

문서를 출력합니다.
HP 프린터 문서를 출력합니다.

Static Method Overriding

- 부모 클래스의 메소드 중에서 정적 메소드를 재정의(오버라이드)하면 부모 클래스 객체에서 호출되느냐 아니면 자식 클래스에서 호출되느냐에 따라서 호출되는 메소드가 달라짐

```
public class Animal {  
    public static void Eat() {  
        Console.WriteLine("Animal의 정적 메소드 Eat()");  
    }  
    public virtual void Sound() {  
        Console.WriteLine("Animal의 인스턴스 메소드 Sound()");  
    }  
}
```

Static Method Overriding

```
public class Cat : Animal {  
    public new static void Eat() { // Cat의 Eat은 Animal의 Eat과 다른 새로운 것  
        Console.WriteLine("Cat의 정적 메소드 Eat()");  
    }  
    public override void Sound() {  
        Console.WriteLine("Cat의 인스턴스 메소드 Sound()");  
    }  
    public static void Main(string[] args) {  
        Cat myCat = new Cat();  
        Cat.Eat(); // Cat의 정적 메소드 Eat()  
        Animal myAnimal = myCat;  
        Animal.Eat(); // Animal의 정적 메소드 Eat()  
        myAnimal.Sound(); // dynamic binding  
    }  
}
```

Cat의 정적 메소드 Eat()
Animal의 정적 메소드 Eat()
Cat의 인스턴스 메소드 Sound()