HCI 프로그래밍

6. Inheritance & Interface

HCI
Human Computer Interaction

Inheritance

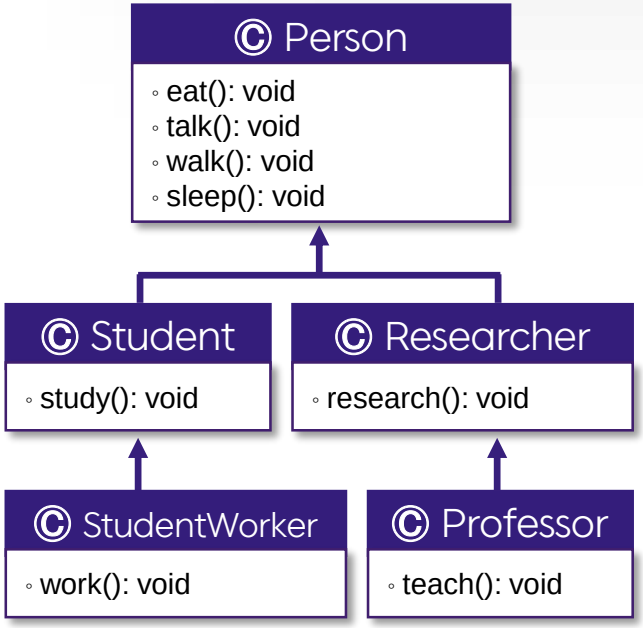
상속 (Inheritance)란 상위 클래스의 기능과 속성을 하위 클래스에게 그대로 물려주는 것을 의미

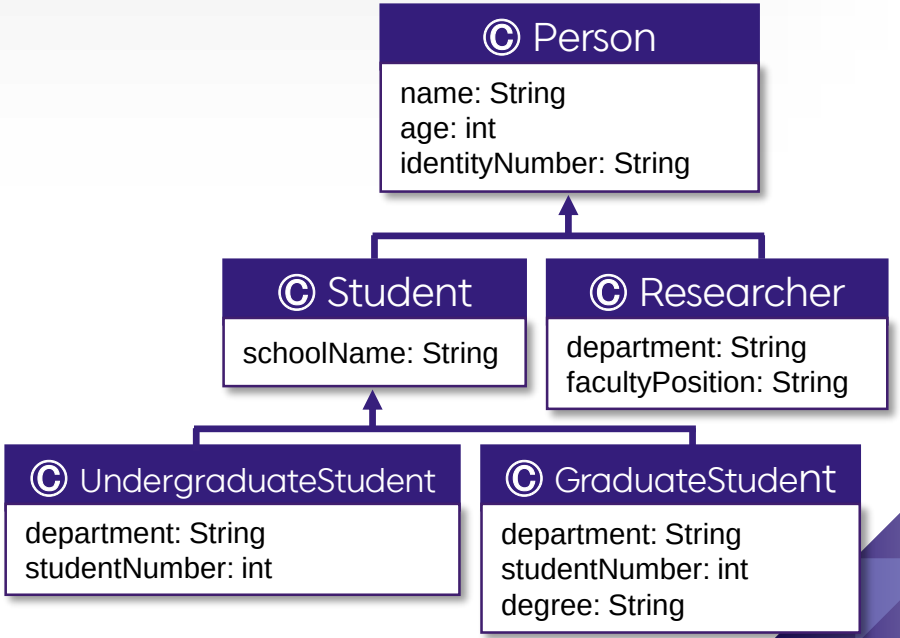
- 상위 클래스를 부모/기반 클래스 (base class),
  하위 클래스를 자식/파생 클래스 (derived class)라고 정의
- 하위 클래스(derived class)는 상속을 받으면
  상위 클래스(base class)의 모든 멤버필드와 메소드를
  사용할 수 있음
- 클래스의 계층 구조 생성
  - ✓ 상속 받은 interface 사용 보장 (upcasting)
- 재사용성(reuse) 및 확장성(extend)
  - ✓ 이미 존재하는 클래스를 구체화(refine)하여 새로운 클래스 생성
  - ✓ 동일한 특성을 재정의할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

Inheritance

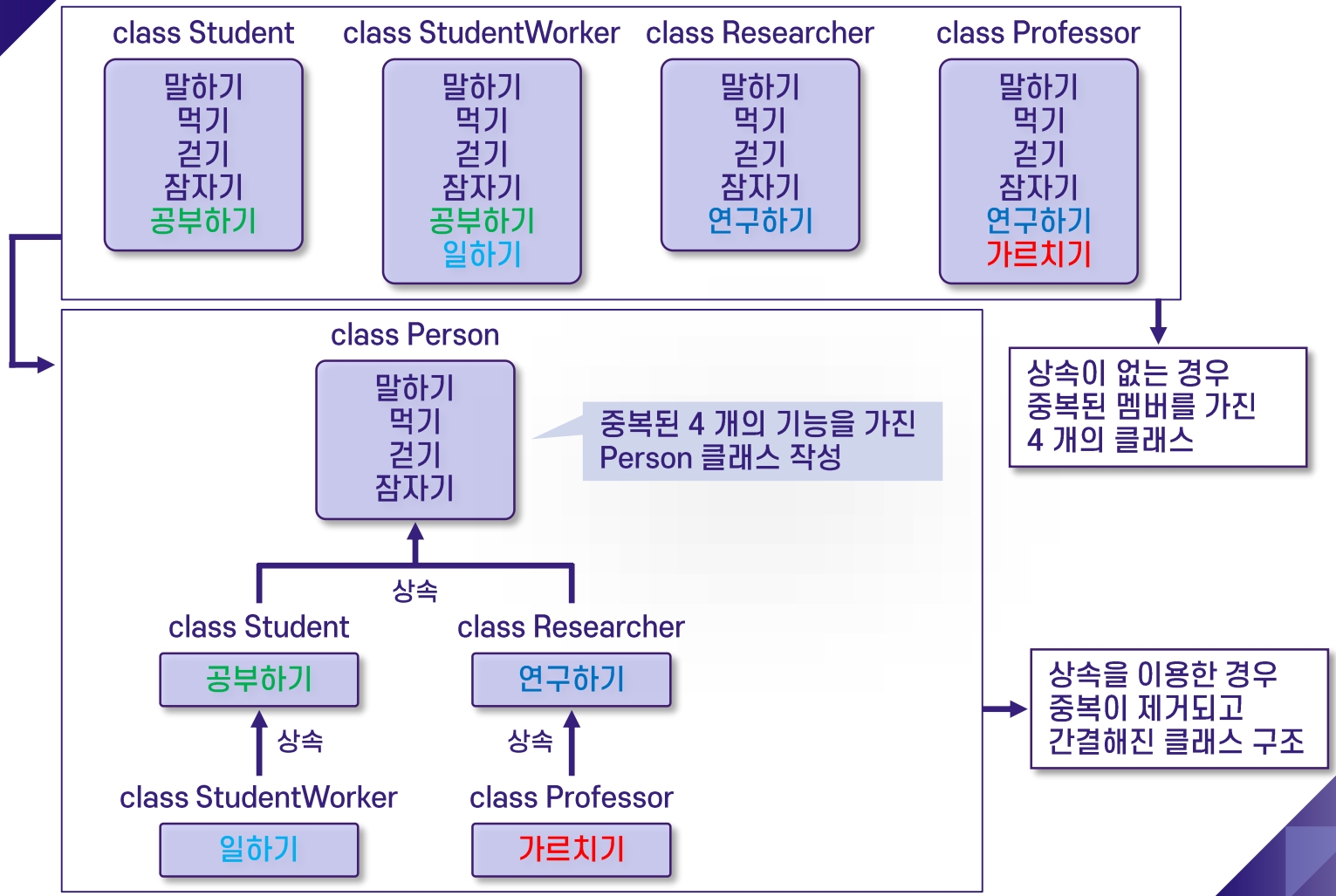
부모 클래스는 한 개 이상의 자식 클래스와 관계를 맺는 계층 구조로 표현 가능

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



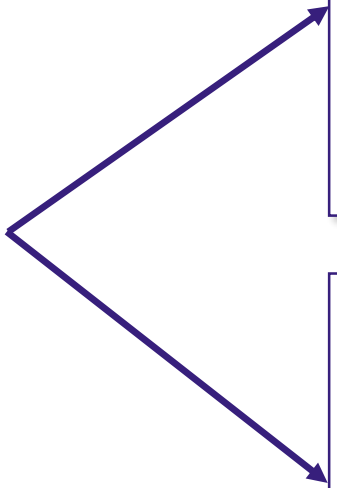


Motivation


class Student
말하기
먹기
걷기
잠자기
공부하기
class StudentWorker
말하기
먹기
걷기
잠자기
공부하기
일하기
class Researcher
말하기
먹기
걷기
잠자기
연구하기
class Professor
말하기
먹기
걷기
잠자기
연구하기
가르치기
상속이 없는 경우
중복된 멤버를 가진
4 개의 클래스
class Person
말하기
먹기
걷기
잠자기
중복된 4 개의 기능을 가진
Person 클래스 작성
상속
class Student
공부하기
class Researcher
연구하기
상속
상속
class StudentWorker
일하기
class Professor
가르치기
상속을 이용한 경우
중복이 제거되고
간결해진 클래스 구조

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
{
    ...
}
```

Inheritance Syntax

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

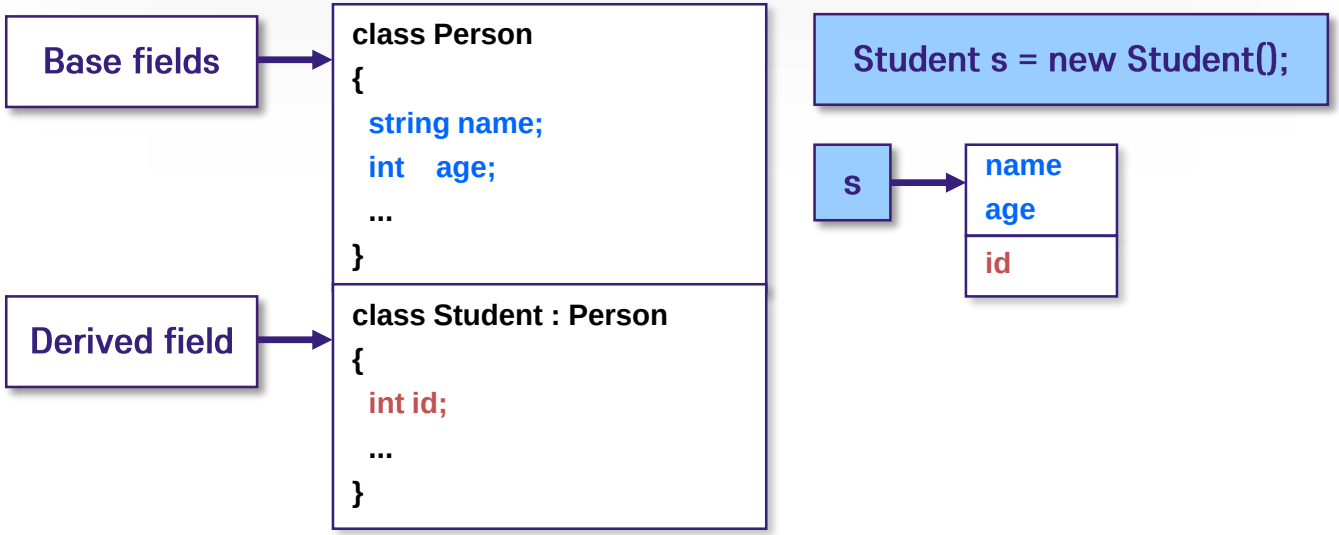
```
class Researcher : Person
{
  string research;
}
```

→ Researcher 에만 있는 멤버를 추가

Inheritance Syntax

상속 시 메모리 할당

- 하위객체 생성시 상위객체도 생성되어 데이터의 메모리 할당



예제

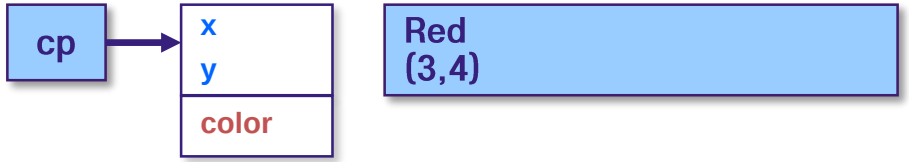
Point & ColorPoint 클래스

```
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
    }
}
```

예제

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 Class

Private 클래스

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

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

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

Member Access

상위 클래스 멤버 접근

- 상위 클래스의 protected와 public으로 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;

Member Access

Student 클래스는 Person 클래스를 상속

- Person 클래스의 private 필드인 age, weight는 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() {
        …
}
```

Member Access

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 Initializer

- 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 Initializer

```
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
```

예제

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 + ")");
    }
}
```

예제

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
    }
}
```

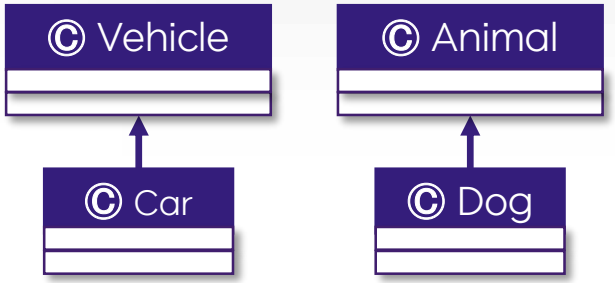
예제

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 관계

- 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).

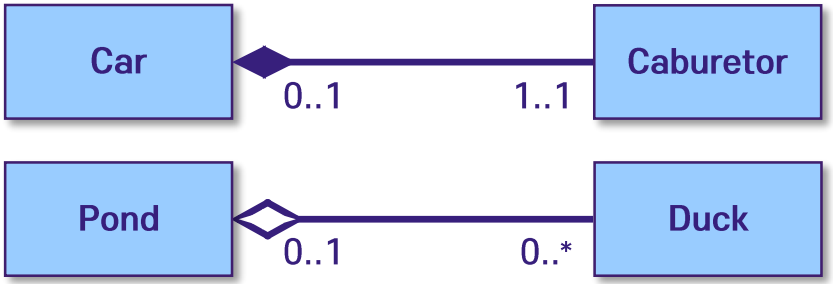


HAS-A 관계

- **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



Has-a Model 예제

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
    } // …
}
```

Has-a Model 예제

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
- Car 클래스는 자식객체의 기능은 잘 모르므로 실제 구현은 위임

```
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 켠다
        c.MusicOn(true);
        // 에어컨 온도를 높인다
        c.TemperatureUp();
    }
}
```

Upcasting

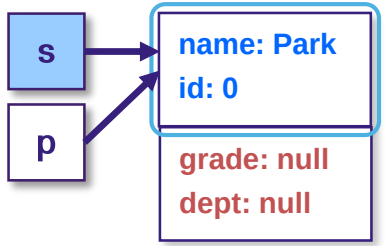
업캐스팅(upcasting)

- 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 가능

```
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
         }
}
```



Downcasting

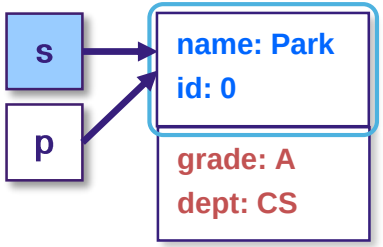
다운캐스팅(downcasting)

- 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
```

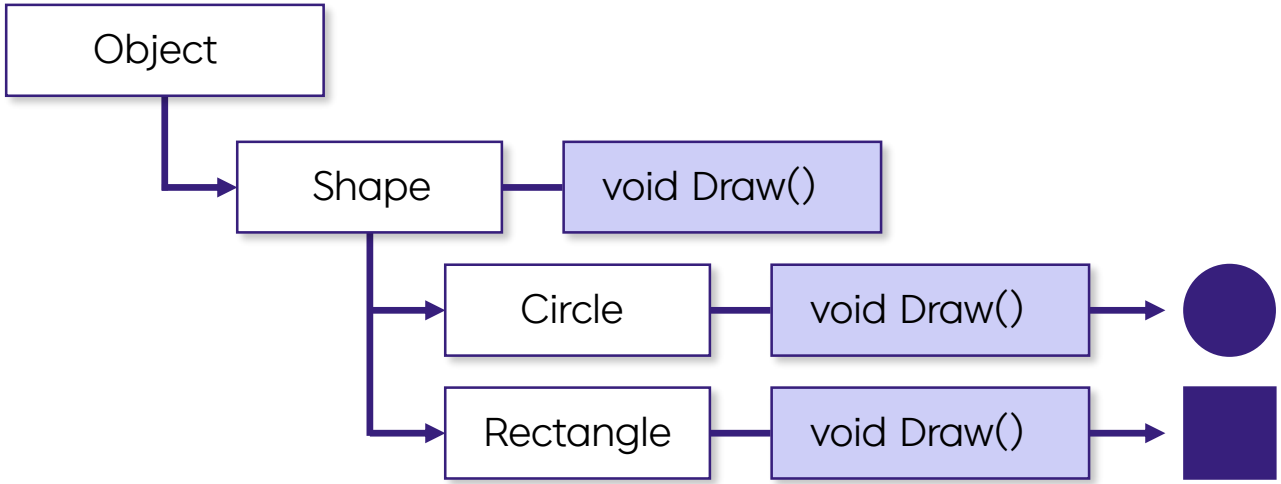
Downcasting 사례

```
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)
- 실행시간에 새로 정의된 Override된 멤버의 내용으로 처리 (Late binding)



Virtual Method

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 method와 override method는 이름, 접근 제한자, 반환 값, 매개변수리스트가 동일해야 함
- static, private을 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

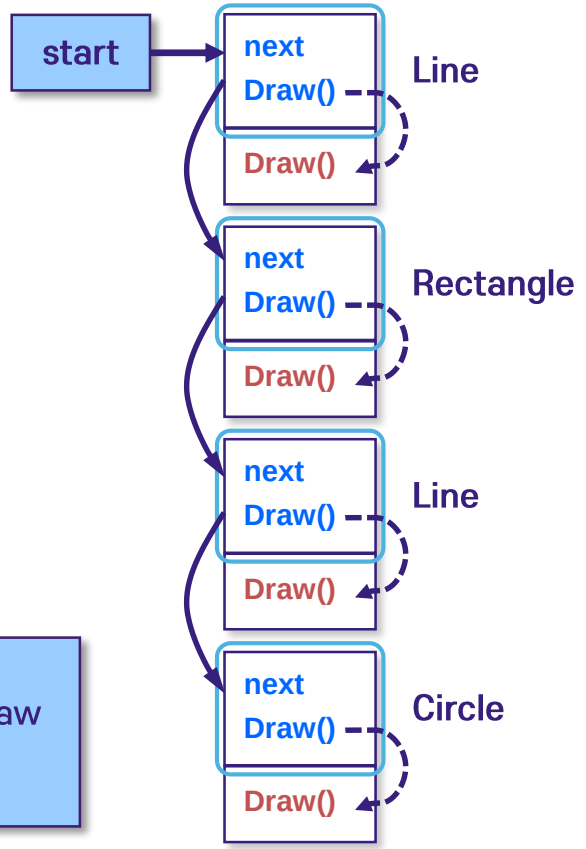
```
public class MethodOverringEx {
        public static void Main(string[] args) {
                Shape s = new Shape();
                Line line = new Line();
                Shape p = new Line(); // upcasting
                Shape r = line; // upcasting
                s.Draw(); // Shape Draw() 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();
        }
}
```

Dynamic binding

Sub Object

Abstract Classes

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

Abstract Method

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.
```

Abstract Class 용도

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
- 각 하위 클래스에서 구체적 행위 구현

계층적 상속 관계를 갖는 클래스 구조를 만들 때

Sealed Class

봉인 클래스 및 봉인 클래스 멤버

- **추가 파생 방지를 위하여 클래스는 자신 또는 멤버를 sealed로 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)이 아니게 됨

```
// 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()

Multiple Inheritance

다중 상속 (Multiple Inheritance)란 하나의 클래스가 여러 개의 클래스로부터 상속을 받는 것

- C#에서는 하나의 클래스가 동시에 2개 이상의 클래스에서 상속 받는 것(Multiple Inheritance)을 지원하지 않음
- **C#에서는 인터페이스(interface)를 이용하여 다중상속 지원**

Interfaces

Interface

- 행동적인 특성(메소드, 속성, 인덱서, 이벤트) 만을 정의
- 인터페이스는 하위 클래스에 구현되어야 하는 기능을 선언할 시 일반적으로 이질적인 클래스들이 공통으로 제공해야 할 메소드들을 선언할 때 사용
- 다중상속을 구현하기 위한 방법으로 사용
- class 대신 interface 키워드를 사용하여 선언

관례적으로 "I" 접두어 사용

public →

```
interface IToken : IBase
{
    int LineNumber( );
    string Name( );
}
```

← *Interface 상속*

인터페이스의 메소드는 구현부분이 없이 '; '으로 처리

Class & Interfaces

클래스/인터페이스 관계

- 클래스 & 클래스 : 상속관계
- 클래스 & 인터페이스 : 구현관계
- 인터페이스 & 인터페이스 : 상속관계

클래스 & 인터페이스 선언 예

```
interface IMyInterface: IBase1, IBase2 {
    void MethodA();
    void MethodB();
}

class ClassA: IFace1, IFace2
{ // class members , implementing interface }

class ClassB: BaseClass, IFace1, IFace2
{ // class members, implementing interface }
```

여러 개의 인터페이스를 상속 받은 인터페이스는 상위 인터페이스의 메소드를 모두 구현해 주어야 함

클래스와 인터페이스를 함께 상속 받는 경우 기본 클래스가 처음에 위치

Class & Interfaces

```
//interface
using System;
interface IScalable {
    void ScaleX(float factor);
    void ScaleY(float factor);
}
public abstract class DrawObject {
    public DrawObject() {}
    public abstract void Print();
}
public class TextObject: DrawObject,
IScalable {
    private string text;
    public TextObject(string text) {
        this.text = text;
    }
    // implemention of IScalable.ScaleX()
    public void ScaleX(float factor) {
        Console.WriteLine("ScaleX: {0} {1}", text,
                          factor);
    }
    // implemention of IScalable.ScaleY()
    public void ScaleY(float factor) {
        Console.WriteLine("ScaleY: {0} {1}", text,
                          factor);
    }
    // implemention of Print()
    public override void Print() {
        Console.WriteLine("TextObject: {0}",
text);
    }
}
```

Class & Interfaces

```
// TextObject 클래스 객체를 array로 처리
class  InterfaceTest {
  public static void Main()  {
    DrawObject[] dArray =
          new  DrawObject[10];

    dArray[0] = new TextObject("Text1");
    dArray[1] = new TextObject("Text2");

 /* array gets initialized here, with classes that
derive from DiagramObject. Some of them
implement IScalable. */
```

```
    foreach (DrawObject d in dArray)
    {
      if (d is IScalable)  {
        IScalable scalable = (IScalable) d;
        scalable.ScaleX(0.1F);
        scalable.ScaleY(10.0F);
        d.Print();
      }
    }
  }
}
```

```
ScaleX: Text1 0.1
ScaleY: Text1 10
TextObject: Text1

ScaleX: Text2 0.1
ScaleY: Text2 10
TextObject: Text2
```

IComparable

IComparable 인터페이스

- 현 객체를 동일한 형식의 다른 객체와 비교하는
  (인터페이스 정렬 순서를 확인하는)
  ComparableTo 메소드를 제공하는 인터페이스
- int CompareTo (Object obj)
  obj 인수는 인터페이스를 구현하는 클래스와 같은 형식
  리턴 값은 현 객체가 obj보다 작으면 0보다 작은 수를, 같으면 0을,
  크면 0보다 큰 수를 리턴하도록 구현

```
public class Age : IComparable {
        public int CompareTo(object obj) {
                if (obj is Age) {
                        Age temp = (Age) obj;
                        return m_value.CompareTo(temp.m_value);
                } throw new ArgumentException("Object is not Age");
        }
        protected int m_value;
}
```

IEquatable

IEquatable 인터페이스

- 현 객체가 동일한 형태의 다른 객체와 같은지 여부를 확인하는 Equals 메소드를 제공하는 인터페이스
- bool Equals (Object obj)
  obj 인수는 인터페이스를 구현하는 클래스와 같은 형식
  리턴 값은 현 객체가 obj과 같으면 true, 다르면 false를 리턴하도록 구현

```
public class Person : IEquatable<Person> {
        public bool Equals(Person p) {
            if (p is Person) {
                return name.Equals(p.name) && age.Equals(p.age);
            } throw new ArgumentException("Object is not Person");
        }
        public string name;
        public int age;
}
```

IEnumerable

IEnumerable 인터페이스

- Collections을 반복하는 열거자 (IEnumerator 객체)를 반환하는 GetEnumerator 메소드를 제공하는 인터페이스
- IEnumerator GetEnumerator ()

IEnumerator 인터페이스

- 컬랙션의 요소들을 참조할 수 있는 아래 3 메소드 및 속성을 갖는 인터페이스
- object Current – 컬랙션의 현재 요소를 가져오는 속성
- bool MoveNext() – 컬랙션의 다음 요소로 이동하는 메소드
- void Reset() – 컬랙션의 첫 번째 요소 앞의 초기 위치로 설정하는 메소드

IEnumerable

```
//IEnumerable interface 구현
using System;
Using System.Collections;

public class Tokens : IEnumerable {
   private string[] elements;
   Tokens(string source, char[] delimiters) {
      elements = source.Split(delimiters);
   }
   // implemention of GetEnumerator()
   public IEnumerator GetEnumerator() {
      return new TokenEnumerator(this);
   }
   // implemention of TokenEnumerator
   private class TokenEnumerator :
IEnumerator {
      private int position = -1;
      private Tokens t;
      public TokenEnumerator(Tokens t) {
         this.t = t;
      }
      public void Reset() {
         position = -1;
      }
      public bool MoveNext() {
         if (position < t.elements.Length – 1) {
            position++;
            return true;
         }
         else {
            return false;
         }
      }
      public object Current {
         get { return t.elements[position]; }
      }
   }
}
```

IEnumerable

```
// Token 테스트
class  EnumeratorTest {
  public static void Main()  {
    Tokens f = new Tokens("This is a sample
                  test.", new char [] {' ', '-'});
    foreach (string item in f) {
      Console.WriteLine(item);
    }
  }
}
```

Interface vs. Abstract Class

인터페이스와 추상클래스의 공통점과 차이점

- 모두 추상의 의미
- new 키워드를 사용하여 객체를 생성하는 것이 불가능
- 파생클래스에서 모든 메서드를 구현하였을 경우 기능을 발휘
- 클래스와 메서드가 abstract로 선언되어 있다면 인터페이스로 변환가능

Interface vs. Abstract Class

변환 예(추상클래스 → 인터페이스)

```
// 추상클래스
abstract public class StarPlayer {
    public abstract void GoodPlay();
    public abstract void Handsome();
}
```

```
// 인터페이스
interface IStarPlayer {
    void GoodPlay();
    void Handsome();
}
```

- 모든 추상클래스가 인터페이스로 될 수 없음
- 추상 메서드는 override 키워드 사용
- 추상 클래스는 다중상속을 지원하지 않음

Interface의 암시적 구현

```
public interface IGunner {
        void Shoot();
}
public interface ISoccerPlayer {
        void Shoot();
}
public class Gunner : IGunner {
        public void Shoot() { Console.WriteLine("총쏘기"); }
}
public class SoccerPlayer : ISoccerPlayer {
        public void Shoot() { Console.WriteLine("공차기"); }
}
// Main () 함수 중간생략
        Gunner g = new Gunner();
        g.Shoot();              // 총쏘기
        SoccerPlayer s = new SoccerPlayer();
        s.Shoot();              // 공차기
```

Interface의 명시적 구현

```
public class GunPlayer : IGunner, ISoccerPlayer {
        void IGunner.Shoot() { Console.WriteLine("총쏘기"); }
        void ISoccerPlayer.Shoot() { Console.WriteLine("공차기"); }
}
// Main () 함수 중간생략
        GunPlayer p = new GunPlayer();
        ((IGunner)p).Shoot();                  // 총쏘기 - IGunner로 형변환
        ((ISoccerPlayer)p).Shoot();            // 공차기 – ISoccerPlayer로 형변환

        // 또는 아래와 같이 해당 객체를 만들고 난 후 참조하도록 하는 방법을 사용
        IGunner g = new GunPlayer();           // IGunner로 형변환
        g.Shoot();                             // 총쏘기
        ISoccerPlayer s = new GunPlayer();     // ISoccerPlayer로 형변환
        s.Shoot();                             // 공차기
```

Default Interface Method

C# 8.0에서 추가된 Default interface method

- 코드 실행 블록을 가지고 있는 메소드로, 자바의 인터페이스 default method와 유사함
- Default interface 메소드를 사용하여 해당 인터페이스의 기존 구현과 소스 또는 호환성을 손상 시키지 않고 이후 버전의 인터페이스에 메서드를 추가할 수 있음
- 클래스는 인터페이스의 멤버를 상속하지 않음

```
interface IA {
   void M() {  Console.WriteLine("IA.M");  } // 기본 인터페이스 메소드
}
class B : IA { } // M() 메소드를 구현 안해도 됨
class C : IA { // default method를 새로 정의
   public void M() {  Console.WriteLine("C.M");  }
}
IA i = new B(); i.M(); // IA.M 기본 인터페이스 메소드 구현 사용
IA j = new C(); j.M(); // C.M
//new B().M(); // error: 클래스 B는 멤버 M()이 없음
new C().M(); // C.M
```

Default Interface Method

Default interface method는 구체적인 재정의가 필요

```
interface IA {
    void M() {  Console.WriteLine("IA.M");   }
}
interface IB : IA {
    void IA.M() {  Console.WriteLine("IB.M");   } // default method override
}
interface IC : IA {
    void IA.M() {  Console.WriteLine("IC.M");   } // default method override
}
interface ID : IB, IC {
}
class D : ID {
    void IA.M() {  Console.WriteLine("D IA.M");   } // default method override
}
D d = new D();
((IA)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((IB)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((IC)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
((ID)d).M(); // 최종 override된 기본 인터페이스 메소드 사용 D IA.M
//d.M(); // error: 클래스 D는 멤버 M()이 없음
```

Default Interface Method

```
interface IA {
    void M() {  Console.WriteLine("IA.M");   }
}
interface IE : IA {
    abstract void IA.M(); // 추상으로 재정의 가능
}
/*class E : IE { // error: IA.M()을 구현해야 함
}*/
class E : IE {
    void IA.M() {  Console.WriteLine("E IA.M");   } // default method override
}
IE ie = new E();
ie.M(); // 최종 override된 기본 인터페이스 메소드 사용 E IA.M
IA ia = new E();
ia.M(); // 최종 override된 기본 인터페이스 메소드 사용 E IA.M
```

Default Interface Method

```
interface IA {
    void M() {  Console.WriteLine("IA.M");   }
}
interface IE : IA {
    abstract void IA.M(); // 추상으로 재정의 가능
}
/*abstract class F : IE { // error: IA.M()을 구현해야 함
}*/
abstract class F : IE {
    void IA.M() {  Console.WriteLine("F IA.M");   } // default method override
}
class G : F {
}
G g = new G();
//g.M(); // error: 클래스 G는 멤버 M()이 없음
//((F)g).M(); // error: 클래스 F는 멤버 M()이 없음
((IE)g).M(); // 최종 override된 기본 인터페이스 메소드 사용 F IA.M
((IA)g).M(); // 최종 override된 기본 인터페이스 메소드 사용 F IA.M
//((IB)g).M(); // run-time error: InvaidCastException g는 IB로 변환될수 없음
```

Base/Derived Conversion :Casting

Upcasting

- 하위 클래스가 상위클래스로의 암시적인 형 변환

Downcasting

- 상위클래스가 하위클래스로의 형 변환 (downcasting) 시 명시적 형 변환 (explicit type conversion) 연산자 필요
- 실행 시 참조변수가 가리키는 실제 객체의 데이터 형 검사
- 실패하는 경우 “InvalidCastException” 발생

```
class UpcastDowncastTest {
   static void Main(string[] args) {
      MyBaseClass c = new MyBaseClass();
      MyDerivedClass d = new MyDerivedClass();
      c = d;                          // upcasting
      d = (MyDerivedClass) c;         // downcasting
   }
}
```

is & as Operator

is 연산자

- 데이터의 형 변환이 가능하면 true 반환

as 연산자

- 객체 사이의 형 변환 연산자
- 오류 발생시 exception 발생 없이 null 반환

```
Bird b;
if (a is Bird)
    b = (Bird) a;   // 안전한 형 변환
else
    Console.WriteLine("Not a Bird");

Bird b = a as Bird; // 형 변환
if (b == null)
    Console.WriteLine("Not a bird");
```

Object Type Conversion

모든 참조형(예, class)은 System.Object으로부터 파생

- 즉, 모든 참조형은 System.Object으로 형 변환이 가능

```
class Bird { … }
class Parrot : Bird { … }
class ObjectTypeConversionTest {
    static void Main(string[] args) {
        Bird b = new Bird();
        Parrot p = new Parrot();
        object o = p;              // object 형 변환
        Bird b2 = o as Bird;       // object형을 Bird 형으로 변환
        if (b2 == null)
            Console.WriteLine(“Object is not a bird");
        else
            Console.WriteLine(“Object is a bird”);
    }
}
```

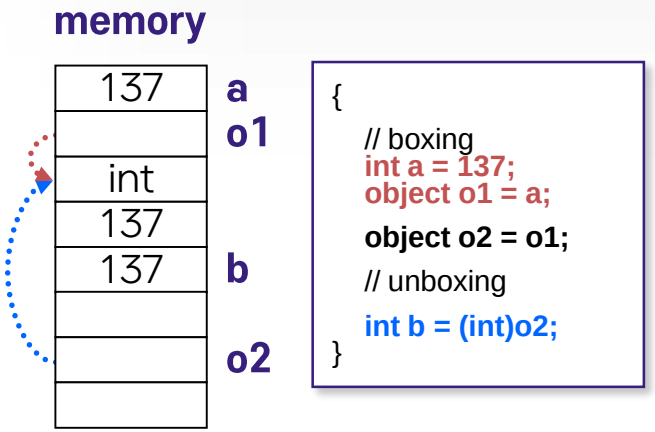
Boxing and Unboxing

Boxing

- 값형 -> 참조형으로 바꾸는 것 (암시적 변환)

Unboxing

- 참조형 -> 값형으로 바꾸는 것 (명시적 변환)



User-Defined Type Conversion

- 클래스나 구조체의 형을 다른 클래스, 구조체 혹은 기본 자료형으로 변환 가능
- 형변환 연산자(conversion operator)를 정의

```
class Sample {
    int number = 42;
    public static implicit operator string (Sample op) {
        return op.ToString();
    } // 문자열로 형변환하는 연산자를 정의했다면 ToString 메소드도 정의해야함
    public override string ToString () {
        return “Object: value=“ + number;
    } // System.Object.ToString 메소드 재정의
}
// 형변환
Sample obj;
string s = obj; // implicit대신 explicit으로 선언되면 string s = (Sample)obj;
```

값 형과 참조 형의 비교

- 값형 경우 ==와 != 연산자를 사용하여 값을 비교
- 참조형 경우 ==와 != 연산자를 사용하여 비교할 수 없음

```
class Point { public int  x; public int y; }
class PointClassComparisonTest {
    static void Main() {
        int i = 1; int j = 1;       // 값형의 비교는 i==j는 true
        Point p1 = new Point();
        Point p2 = new Point();
        Point p3 = p1;                    // 같은 객체를 참조하므로 p1==p3는 true
        p1.x = 1; p1.y = 1;
        p2.x = 1; p2.y = 1;
        if (p1 == p2)              // 참조형의 비교는 p1==p2는 false
                Console.WriteLine("p1과 p2가 같다");
        else
                Console.WriteLine("p1과 p2가 다르다");
    }
}
```

Equals & Equality Operator

참조형 경우 비교 연산자를 사용할 수 있도록 Equals(..) overriding과 == operator와 != operator를 overloading함

```
class Point: IEquatable<Point> {
    public int  x; public int y;
    public Point(): this(0, 0) {}
    public Point(int x, int y) { this.x = x; int.y = y; }
    // operator== overload
    public static bool operator==(Point p1, Point p2) { return p1.Equals(p2); }
    // operator!= overload
    public static bool operator!=(Point p1, Point p2) { return !p1.Equals(p2); }
    public override int GetHashCode() { return x ^ y; }
    public override bool Equals(object obj)
{
        if (!(obj is Point))
            return false;
        return Equals((Point)obj);
    }
```

Equals & Equality Operator Overloading

```
// IEquatable
    public bool Equals(Point other)
    {
        if (this.x == other.x && this.y == other.y)
            return true;
        return false;
    }
} // end of Point class
class PointClassComparisonTest {
    static void Main(string[] args) {
        Point p1 = new Point(3, 4);
        Point p2 = new Point(3, 4);
        Point p3 = p1;
        if (p1 == p2)     // Equals와 ==operator로 p1==p2는 true
            Console.WriteLine("p1과 p2가 같다");
        if (p1 == p3)     // 같은 객체를 참조하므로 p1==p3는 true
            Console.WriteLine("p1과 p3가 같다");
    }
}
```

Method Overloading

메소드 오버로딩(Overloading)

- 한 클래스 내에서 두 개 이상의 이름이 같은 메소드 작성
  - ✓ 메소드 이름이 동일하여야 함
  - ✓ 매개 변수의 개수가 서로 다르거나, 타입이 서로 달라야 함
  - ✓ 리턴 타입은 오버로딩과 관련 없음

```
// 메소드 오버로딩이 성공한 사례
class MethodOverloading {
        public int Sum(int i, int j) {
                return i + j;
        }
        public int Sum(int i, int j, int k) {
                return i + j + k;
        }
        public double Sum(double i, double j) {
                return i + j;
        }
}
```

```
// 메소드 오버로딩이 실패한 사례
class MethodOverloadingFail {
        public int Sum(int i, int j) {
                return i + j;
        }
        public double Sum(int i, int j) {
                return (double)(i + j);
        }
}
```

Method Overriding

메소드 오버라이딩(Method Overriding)

- 슈퍼 클래스의 메소드를 서브 클래스에서 재정의
  - ✓ 슈퍼 클래스의 메소드 이름, 메소드 인자 타입 및 개수, 리턴 타입 등 모든 것 동일하게 작성
    - **이 중 하나라도 다르면 메소드 오버라이딩 실패**
- 동적 바인딩 발생
  - ✓ 서브 클래스에 오버라이딩된 메소드가 무조건 실행되도록 동적 바인딩 됨