객체지향설계
Fall 2018

임성수 교수

Lecture 3: Introduction to Objects

수업 내용

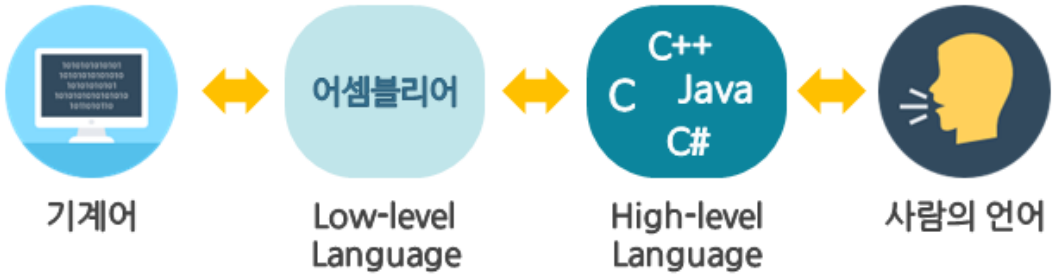
1. 추상화 과정
2. 객체와 인터페이스
3. 구현
4. 상속
5. 다형성
6. 객체의 생성과 소멸
7. 예외 처리
8. 분석과 설계
9. 익스트림 프로그래밍 (XP)
10. 왜 C++인가

1. 추상화 과정

모든 프로그래밍 언어는 **추상화(abstraction)**를 제공

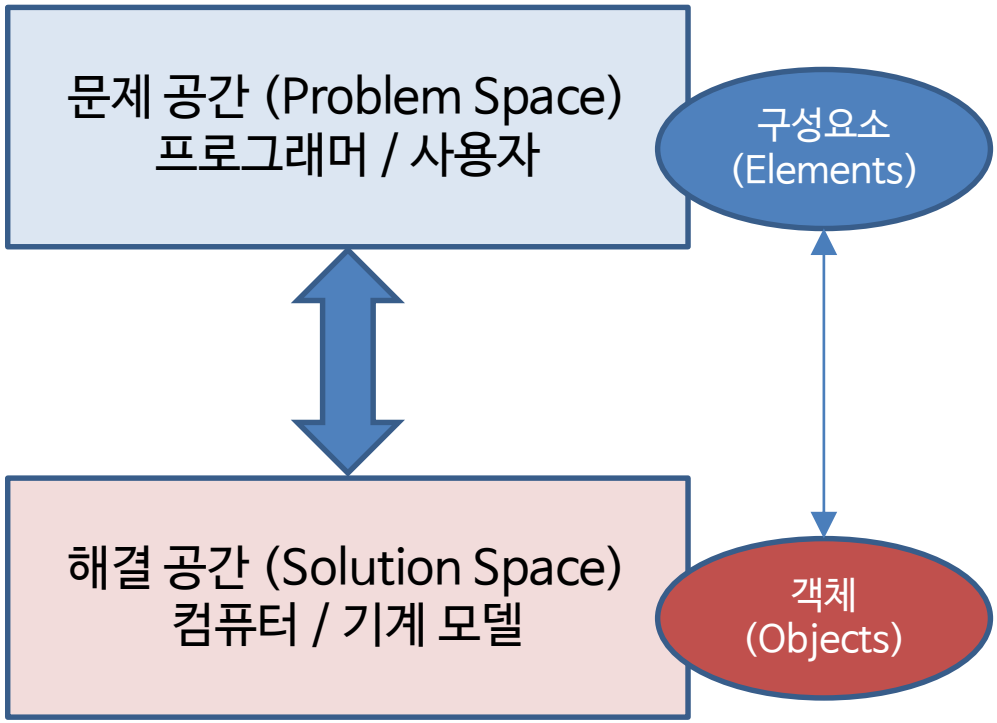
프로그래밍 언어의 추상화

- 초기 고급 언어(Fortran, BASIC, C)는 저급 언어(어셈블리어)의 추상화
- 추상화의 정도는 프로그래밍 언어의 수준 정의
- 저급 언어는 기계어와 거의 1:1 대응되는 간단한 추상화
- 고급 언어는 사람의 언어와 기계어 사이의 매핑(mapping) 고려



1. 추상화 과정

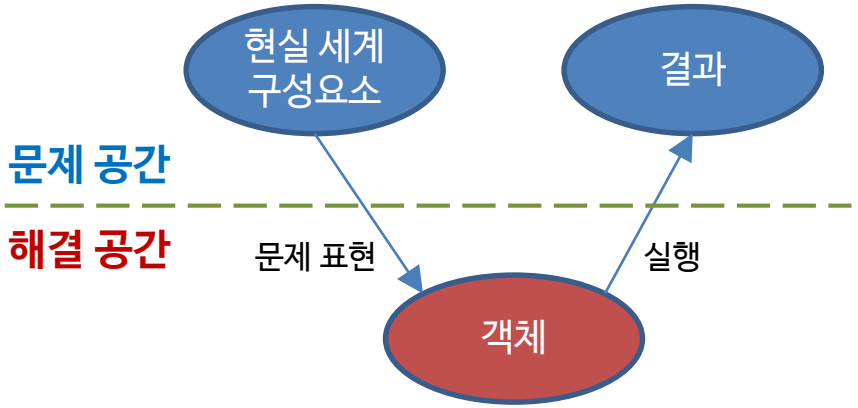
추상화 과정: 문제 공간과 해결 공간 사이의 매핑



1. 추상화 과정

객체지향 방식

- 프로그래머에게 문제 공간의 구성요소들을 표현 가능한 툴 제공
  - **객체: 해결 공간에서 표현된 문제 공간의 구성요소**
- 절차지향: 문제로부터 정답을 얻어내는 과정을 표현
- 객체지향: 문제의 각 구성요소 및 관계를 표현
- 문제를 효율적으로 풀 수 있음
- 유지/보수 및 확장에 유리



1. 추상화 과정

Smalltalk: C++, Java의 기초가 된 객체지향 언어

- 객체지향 프로그래밍의 기본 노선 정립 (1970s)

Smalltalk의 기본 특성

1. 모든 것은 **객체**
2. 프로그램은 객체 사이의 **메시지** 전달을 이용하여 통신
3. 각각의 객체는 자신만의 메모리를 가짐
4. 모든 객체는 **형(type)**을 가짐
5. 동일한 형의 객체는 동일한 메시지를 사용

2. 객체와 인터페이스

클래스

- 초기 객체지향 언어 Simula-67에서 사용됨 (1960s)
  - Smalltalk 등 이후 객체지향 언어 개발에 영향을 줌
- **자료형**(data type)의 일종
  - **상태**(characteristic, 구성요소), **행위**(behavior, 기능)의 집합
  - 필요에 따라 새로운 자료형을 정의 가능

객체

- 클래스로부터 생성된 인스턴스(instance)
- 문제 내의 구성요소처럼 조작 가능
- 메시지를 요청하거나 받을 수 있음

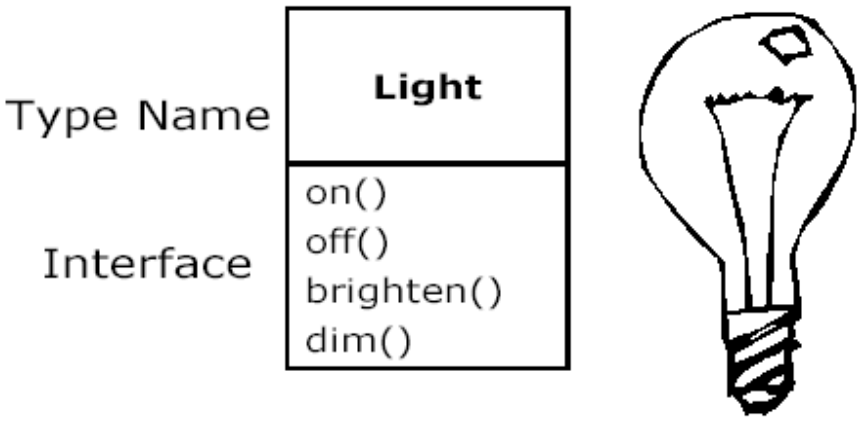
2. 객체와 인터페이스

인터페이스

- 객체의 요구사항은 그들의 인터페이스(interface)에 의해 정의됨

[예제] 전구

```
Light lt;
lt.on();
```



3. 구현 – 은폐

접근 제어

- 버그를 줄이기 위함
- 부주의나 잘못된 변경에 의해 쉽게 변조되지 않도록 방지
- 클래스 설계자(class creator):
  필요한 부분만 노출시키고 나머지는 숨김
- 클라이언트 프로그래머(client programmer):
  어플리케이션 개발을 위해 클래스의 접근 및 활용

C++ 접근 제어

- 접근 제어 지시자를 통해 관리(public, private, protected)

3. 구현 – 은폐

[예제] 접근 제어
(access control)

```
class A {
private:
    int data=40;
public:
    void msg() {cout << "Hello C++" << endl;}
};

void main(){
   A* obj;
   obj = new A();
   cout << obj.data << endl; // Compile Error
   obj->msg(); // OK
}
```

C++

```
class A {
private int data=40;
public void msg(){ System.out.println("Hello Java"); }
}

public class Simple {
 public static void main(String args[]) {
   A obj=new A();
   System.out.println(obj.data); // Compile Error
   obj.msg(); // OK
   }
}
```

Java

실행 결과

```
Hello C++
```

3. 구현 – 클래스 재사용

코드의 재사용

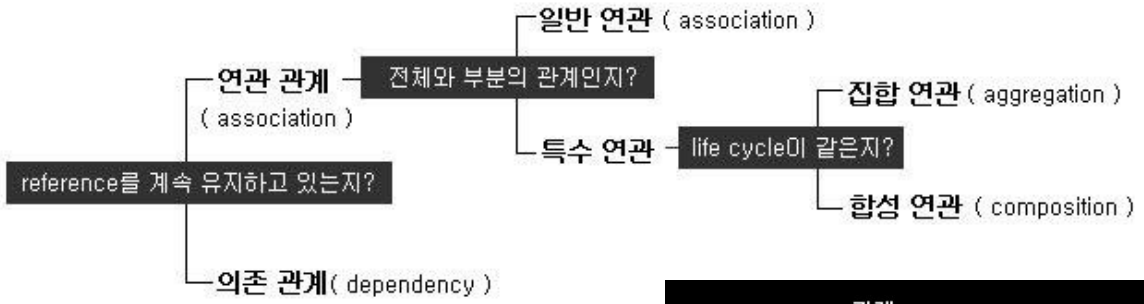
- **재사용(reuse)**: 객체지향 언어가 제공하는 최대의 이점 중 하나

클래스의 재사용

- 해당 클래스의 객체를 직접 사용할 수 있음
- 해당 클래스의 객체를 새로운 클래스의 객체로 위치시킬 수 있음
  - 연관 관계(association): 한 객체가 다른 객체와 연결되어 있음
  - 집합 연관(aggregation): 연관 관계 + 전체/부분 관계
  - 복합 연관(composition): 연관 관계 + 전체/부분 관계
    복합 객체가 부분 객체를 생성 및 삭제

3. 구현 – 클래스 재사용

[예제] UML 클래스 다이어그램



클래스의 관계
의존: use a 관계
연관: has a 관계
상속: is a 관계 (일반화)

Member Access
public (+)
private (-)
protected (#)

관계	UML 표기
Generalization (일반화)	
Realization (실체화)	
Dependency (의존)	
Association (연관)	
Directed Association (직접연관)	
Aggregation (집합, 집합연관)	
Composition (합성, 복합연관)	

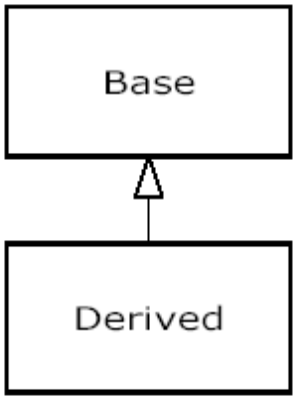
4. 상속 – 인터페이스 재사용

재사용

- 새로운 클래스를 만들 때 연관 관계를 통한 **재사용을 우선 고려**
- 비슷한 기능을 하는 새로운 것을 만들 때 인터페이스 재사용을 고려

상속(Inheritance)

- **인터페이스의 재사용**
- 이미 정의된 클래스를 바탕으로 필요한 기능 추가/수정
- 일반화 관계
  - 기존(base) 클래스: 슈퍼 클래스, 부모 클래스
  - 파생(derived) 클래스: 서브 클래스, 자식 클래스



4. 상속

[예제] 상속(inheritance)

```java
class Calc {
    int z;
    public void add (int x, int y) { z = x + y; }
}
public class My_Calc extends Calc {
    public void mul (int x, int y) { z = x * y; }
    public static void main(String args[]) {
        int a = 20, b = 10;
        My_Calc demo = new My_Calc ();
        demo.add (a, b);
        demo.mul (a, b);
    }
}
```

Java

```cpp
class Calc {
protected:
    int z;
public:
    void add(int x, int y) {z = x + y;
    cout << z << endl;}
};
class My_Calc : public Calc {
public:
    void mul (int x, int y) {z = x * y;
    cout << z << endl;}
};
int main() {
    int a = 20, b = 10;
    My_Calc demo;
    demo.add (a, b);
    demo.mul (a, b);
}
```

C++

실행 결과

```
30
200
```

4. 상속

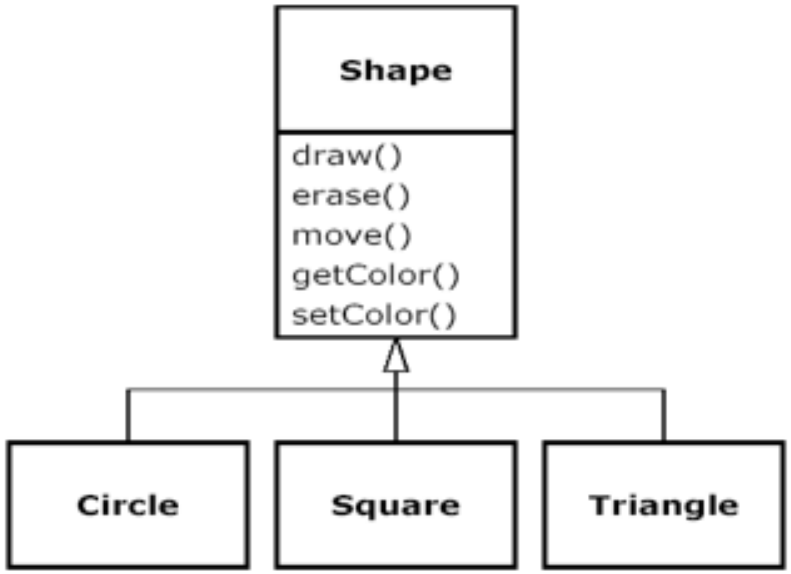
[예제]

Base class

- type: Shape
- 상태: size, color, position 등
- 행위: draw(), erase(), move() 등

Derived classes (inherited)

- types: Circle, Square, Triangle 등
- shape 클래스를 기반으로
  각자의 추가적인 상태, 행위 정의 가능

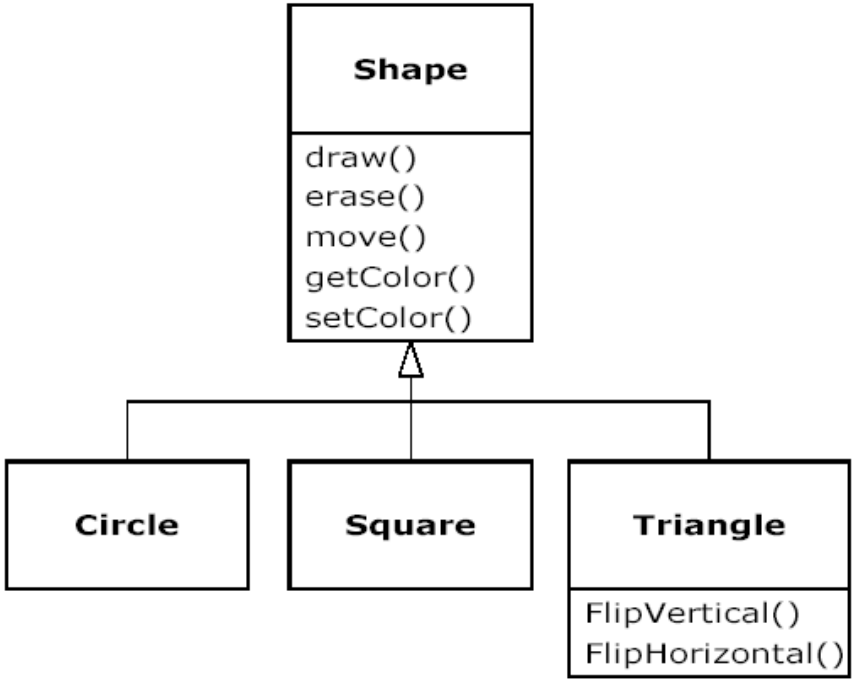


4. 상속

[예제] (cont'd)

Derived class

- type: Triangle
- 추가적인 상태, 행위 정의 가능
- x축 대칭, y축 대칭 등

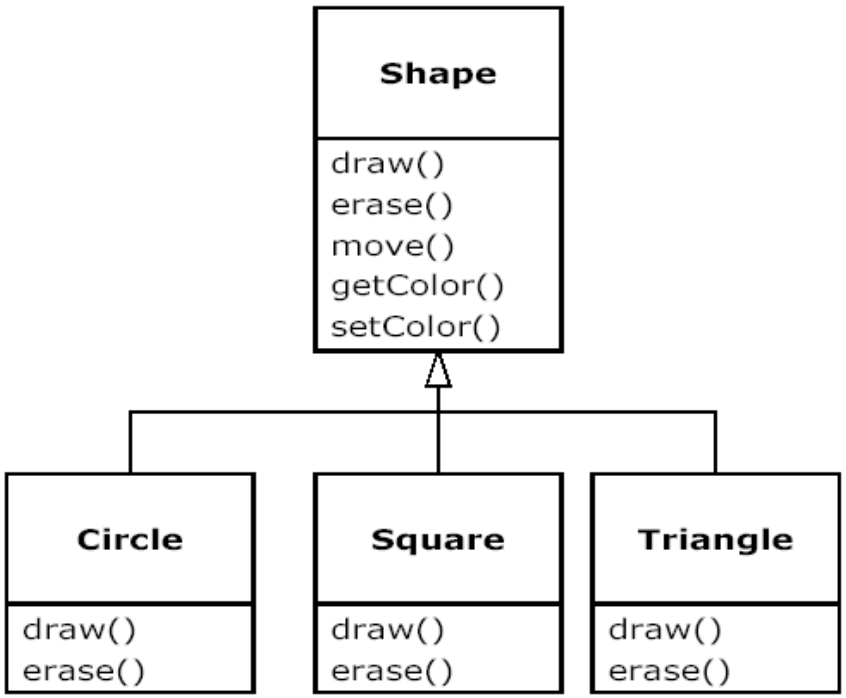


4. 상속

[예제] (cont'd)

Derived classes

- types: Circle, Square, Triangle 등
- shape 클래스에서 정의된
  행위를 바꿔줄 수 있음
- **오버라이딩** (Overriding, 재정의)
  예제: 원, 정사각형, 삼각형의
  형태에 따라서 작도법이 다양함



4. 상속

is-a 관계와 is-like-a 관계

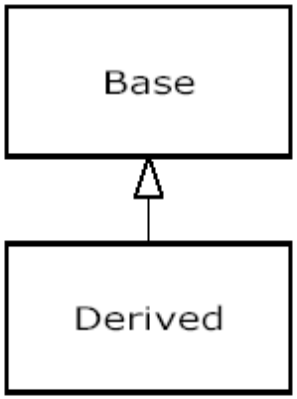
- 기반 클래스를 파생 클래스로 **대체(치환)** 가능
- 리스코프 치환 원칙: 자료형 D가 자료형 B의 하위형이라면 프로그램 속성의 변경 없이 자료형 B의 객체를 자료형 D의 객체로 대체(치환)할 수 있음
  (Liskov)

[예제]

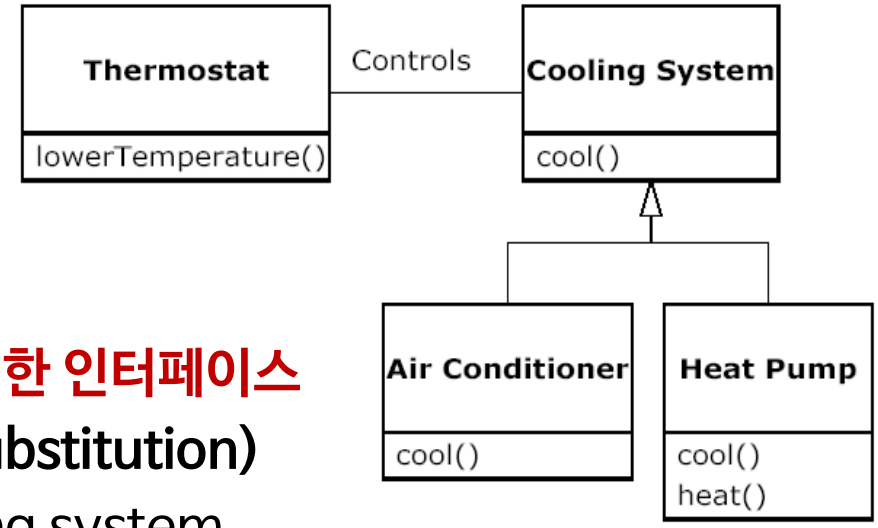
Base: Shape

Derived: Circle

```
Shape* s = new Circle(); // OK
Circle* c = new Shape(); // error
```



4. 상속



is-a 관계

- 파생 클래스가 기존 클래스와 **동일한 인터페이스**
- 기존 클래스를 **순수 대체(pure substitution)**

[예제] Air conditioner **is-a** cooling system.

Is-like-a 관계

- 파생 클래스가 기존 클래스와 **다른 메소드를 추가**로 가짐
- 기존 클래스를 대체 가능하지만 100% 같진 않음

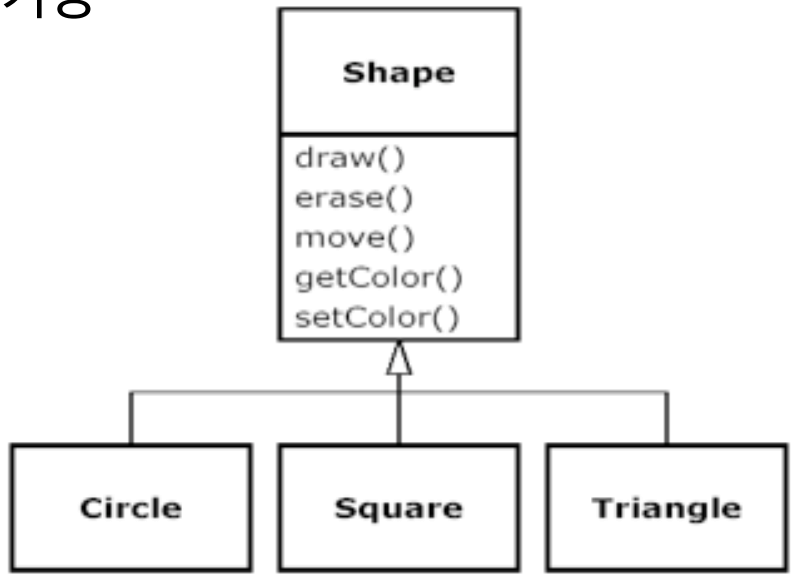
[예제] Heat pump **is-like-a** cooling system.

5. 다형성

다형성(Polymorphism)

- 객체들의 type이 다르면 **동일한 메시지에 대해 다른 동작/결과**
- 메시지를 보내는 측에서 객체의 type을 몰라도 됨
- 객체의 type에 따라 자동적으로 적합한 동작 결정
- 특정 타입에 의존하지 않은 코드 작성 가능

[예제]
Shape클래스의 draw()를 통해
Circle, Square, Triangle 등에서
서로 다른 방식으로 작도 가능



5. 다형성

오버라이딩(Overriding)

- 기존 클래스의 메소드를 파생 클래스에서 재정의

```
#include "pch.h"
#include <iostream>
using namespace std;

class Animal
{
public:
  void Crying()
  {
    cout << "엉엉" << endl;
  }
};
```

```
class Cat : public Animal
{ };
class Dog : public Animal
{ };

int main()
{
  Cat cat;
  cat.Crying();
  Dog dog;
  dog.Crying();
}
```

실행 결과

```
엉엉
엉엉
```

5. 다형성

오버라이딩(Overriding)

- 기존 클래스의 메소드를 파생 클래스에서 재정의

```
class Cat : public Animal
{
public:
  void Crying()
  {
    cout << "야옹" << endl;
  }
};
```

```
class Dog : public Animal
{
  public:
  void Crying()
  {
    cout << "멍멍" << endl;
  }
};
```

실행 결과

```
야옹
멍멍
```

5. 다형성

바인딩(Binding)

- 함수 호출을 함수의 몸체와 연결하는 것
- **정적 바인딩**(early-/static- binding)
  컴파일(compile)시 호출되는 함수 결정, 실행 속도 빠름
- **동적 바인딩**(late-/dynamic- binding)
  실행(runtime)시 호출되는 함수 결정, 가상 함수 사용

[예제] 정적 바인딩: 컴파일 시 타입을 확정 지어서 Animal의 Crying을 불러옴

```
Int main()
{
  Animal* cat = new Cat();
  cat->Crying();
};
```

실행 결과 **엉엉**

5. 다형성

가상 함수(Virtual function)

- 기존 클래스에서 **virtual** 키워드를 붙여서 정의하면 가상 함수가 됨
- 가상화된 멤버 함수를 오버라이딩하면 동적 바인딩 가능

[예제] 동적 바인딩: 호출 시 오버라이딩된 함수가 있는지 확인 후 실행

```
class Animal
{
public:
  virtual void Crying()
  {
    cout << "엉엉" << endl;
  }
};
```

```
Int main()
{
  Animal* cat = new Cat();
  cat->Crying();
};
```

실행 결과 **야옹**

5. 다형성

추상 클래스(Abstract class)

- **순수 가상 함수**(Pure virtual function)
  기능을 구현하지 않은 가상화된 멤버 함수
- **추상 클래스**(Abstract class):
  가상화된 멤버 함수를 가진 클래스
- 일반(concrete) 클래스와 달리 객체를 인스턴스로 갖지 못함
- 파생 클래스를 통해 오버라이딩해야 일반 클래스가 될 수 있음
- 추상적인 형태만 제안하고, 실제 구현은 파생 클래스에서 이뤄짐

```
class Animal
{
  public:
  virtual void Crying() = 0;
};
```

컴파일 에러

'Animal': 추상 클래스를
인스턴스화할 수 없습니다.

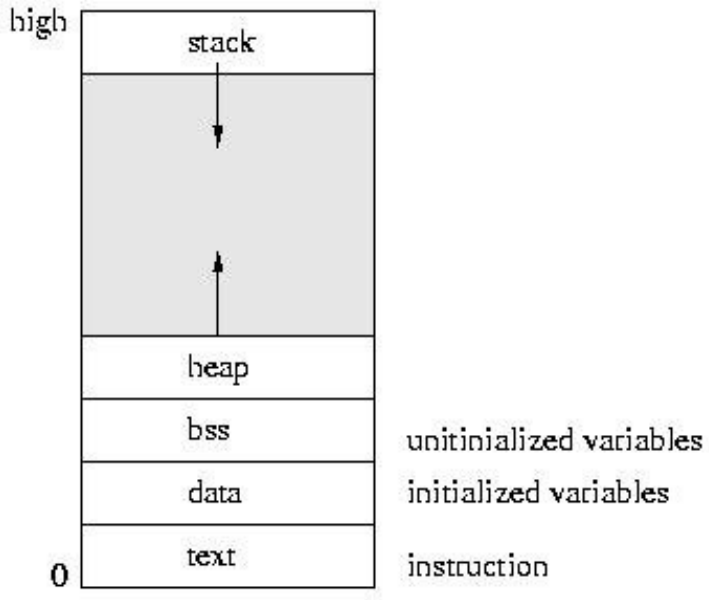
6. 객체의 생성과 소멸

정적 할당

- 미리 메모리 공간을 할당 받고 실행
- 실행 도중 소멸되지 않고 종료 시 알아서 회수

동적 할당

- 상황에 따라 **원하는 크기만큼 할당/반환**
- 할당 후에도 크기 조정 가능
- 사용 후 명시적으로 해제해야 함



7. 예외 처리

에러 처리

- 에러 처리는 가장 복잡한 이슈 중 하나
- 좋은 에러 처리 방식을 설계하기 어려움

예외 처리

- 예외 처리는 프로그래밍 언어와 때때로 운영 체제에서 바로 연결
- 예외를 클래스로 정의하고 **예외 객체**를 만들 수 있음
  - try: 예외 발생이 예상되는 코드 작성
  - catch: 예외를 처리하는 코드 작성
  - throw: 예외가 발생했음을 알림

7. 예외 처리

[예제]
try
catch
throw

```
#include "pch.h"
#include <iostream>
using namespace std;

int main() {
  int num1, num2;
  cout << "두 개의 숫자 입력 : ";
  cin >> num1 >> num2;

  try {
    if (num2 == 0)
      throw num2;
    cout << "몫 : " << num1 / num2 << endl;
    cout << "나머지 : " << num1 % num2 << endl;
  }
  catch (int expn) {
    cout << "제수는 " << expn << "이 될 수 없습니다." << endl;
    cout << "프로그램을 다시 실행하세요." << endl;
  }
  return 0;
}
```

8. 분석과 설계

모델링

- **분석과 설계**: 구현 및 테스트의 이전 단계
- 과도한 설계와 분석 마비(analysis paralysis)를 지양해야 함
- 분석과 설계를 빠르게 수행하고 시스템의 테스트를 구현해야 함

객체와 인터페이스

- 객체와 인터페이스를 정하고 프로그램 작성 시작
  - 객체: 시스템에 포함될 구성요소
  - 인터페이스: 객체 사이에서 보낼 수 있는 메시지

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

0. Make a plan

절차 설계 및 **요구사항 분석**

- 프로세스 구현을 위해 어떤 절차가 필요한지 결정

계획 없이 바로 코딩을 시작하는 것도 좋음

- 이미 잘 이해하고 있는 문제인 경우 더 적합할 수 있음

프로젝트를 작은 단위씩 쪼개서 해결

- 문제를 덜 어렵게 느끼고 부분적인 성취 가능

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

1. What are we making?

이전 단계의 요구사항 분석

- 사용자가 원하는 바를 탐색 → 이를 바탕으로 시스템 명세 작성

시스템 명세(System specification)

- 요구사항 분석의 결과물
- 프로그램이 **'어떤' 동작을 할 것인지** 설명 ('어떻게'는 이후 단계)
- 문제에 대한 최상위 레벨 탐색
- 리스트화 및 다이어그램 작성

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

1. What are we making?

사용 사례 (Use case)

- 시스템이 지니는 주요 특성(key feature)을 파악
- 다음의 질문들에 대한 답을 설명
  - 누가 시스템을 사용하는가? (행위자)
  - 행위자들이 시스템에서 무엇을 할 수 있나? (시스템 역할, 동작)
  - 행위자들과 동작들 사이의 다양한 변수가 있나? (변형 파악)
  - 어떠한 문제점들이 발생하는가? (문제점, 예외 파악)

8. 분석과 설계

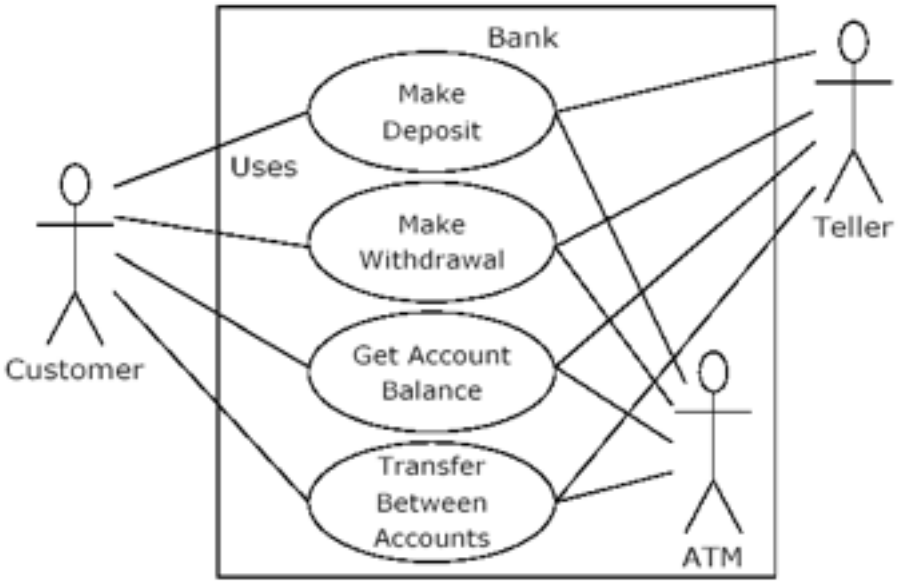
소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

1. What are we making?

사용 사례 다이어그램 (Use case diagram)

- 사람: **행위자**
- 박스: **시스템**의 경계
- 타원: **사용 사례**
- 선: 행위자와 사용 사례 사이의 **상호작용**



8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

2. How will we build it?

Class-Responsibility-Collaboration (CRC) 카드

- 객체지향 설계에서 사용되는 브레인 스토밍 툴
- **클래스**와 **상호작용**을 결정
- 각 카드는 하나의 클래스를 표현하고 각 카드의 빈 칸들을 채워나감
- 카드의 구성
  - 클래스의 이름 (Class name): 클래스를 대표하는 이름 선정
  - 클래스의 책임 (Responsibility): 멤버 함수의 이름 및 기능 작성
  - 클래스의 협력 (Collaboration): 연관 있는 다른 클래스를 작성

8. 분석과 설계

2. How will we build it?

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

[예제] CRC – index card
클래스의 이름, 책임, 협력 사항을 기재

class name

responsibility

collaboration

Order	
Check if items in stock	*Order*
Determine price	*Customer*
Check for valid payment	
Dispatch to delivery address	

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

2. How will we build it?

CRC 카드를 작성하는 이유

- 클래스에 대해 작은 카드에 모두 요약할 수 없다면 복잡해질 것
- 필요한 경우 하나의 클래스를 여럿으로 나누는 걸 고려
- 이상적인 클래스는 **한 눈에 이해가 되어야 함**

CRC 카드를 작성한 후 UML 등 설계 도구를 통해 그려봄

- 클래스 다이어그램 작성

8. 분석과 설계

2. How will we build it?

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

객체 설계의 5단계

1. Object discovery: 어떠한 기초 클래스들이 있어야 할지 고려
2. Object assembly: 객체를 개발할 때 필요한 멤버들 탐색
3. System construction: 객체 간의 필요한 통신 및 관계 고려
   클래스를 변경하거나 새로 설계할 수 있음
4. System extension: 기존 설계가 시스템 확장에 적합한지 고려
   시스템 구성을 바꾸거나 클래스 계층을 바꿀 수 있음
5. Object reuse: 새로운 프로그램, 상황에서 객체의 재사용을 고려
   불필요한 객체의 생성을 줄일 수 있음

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

2. How will we build it?

클래스를 만들 때 고려할 점

1. 특정 문제를 해결하는 클래스를 생성하고 다른 문제에 적용 가능하게 확장
2. 시스템 설계의 주요 사항은 **필요한 클래스들을 탐색**하는 것
3. 처음부터 모든 걸 알아야 한다고 생각하지 말 것
4. 프로그래밍을 시작: 나쁜 코드를 두려워하지 말고 구현 가능 여부를 따짐
5. 항상 간단하게: **크고 복잡한 인터페이스보다 작아도 확실한** 객체가 더 유용

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

3. Build the core

한 번에 가능한 프로세스가 아님

- **시스템의 반복적인 개발**

시스템 구조의 핵심을 찾아야 함

- 시스템 실행을 위해 구현해야 할 부분을 찾음

향후 추가 개발할 수 있는 뼈대를 구축

- 실현 가능성을 고려한 **프레임워크 생성**

8. 분석과 설계

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

4. Iterate the use cases

반복적인 사용 사례 분석

- 사용 사례: 개발한 기능과 관련된 사용 사례를 계속 분석
- 피드백을 통해 더 좋은 아이디어를 얻음
- **유효성(validation)**을 높일 수 있음
- **변화하는 요구사항**에 대응할 수 있음

반복의 중단

- 목표로 하는 기능을 성취했을 때
- 데드라인이 끝났을 때
- 사용자가 현 버전에 만족할 때

8. 분석과 설계

5. Evolution

유지/보수

- 사용자가 필요로 하는 다른 기능 추가
- 버그가 발생할 경우 고침
- 수정하는 것에 두려워할 필요 없음

소프트웨어 개발 5단계

0. Make a plan
1. What are we making?
2. How will we build it?
3. Build the core
4. Iterate the use cases
5. Evolution

9. 익스트림 프로그래밍 (XP)

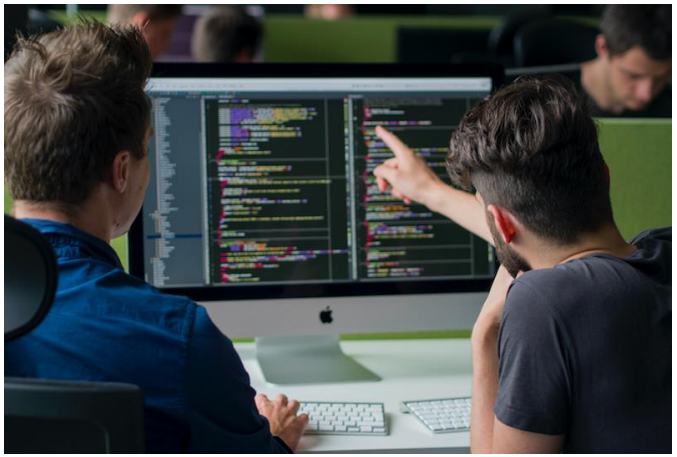
익스트림 프로그래밍 (Extreme Programming, XP)

- 프로그래밍 작업의 철학
- 프로그래밍을 위한 가이드라인
- 방법
  - 테스트 코드를 먼저 만든다.
  - **테스트를 기반으로 프로젝트를 완성**시켜 나간다.
  - 반복적인 프로토타입 제공으로 변화에 민첩하게 대응한다.
  - **페어 프로그래밍 (Pair programming)**:
    팀을 편성하고 모든 사람이 코드를 알 수 있게 돌아가며 작업

9. 익스트림 프로그래밍 (XP)

페어 프로그래밍 (Pair Programming)

- 애자일 개발 방법론 중 하나
- 작업공간마다 팀을 이뤄 코드 작성
  - **한 명이 생각하고 한 명이 실제 코드 작성**
  - 역할을 번갈아 가며 수행
- 시간이 더 걸리지만 결함이 감소
- 장/단점이 존재

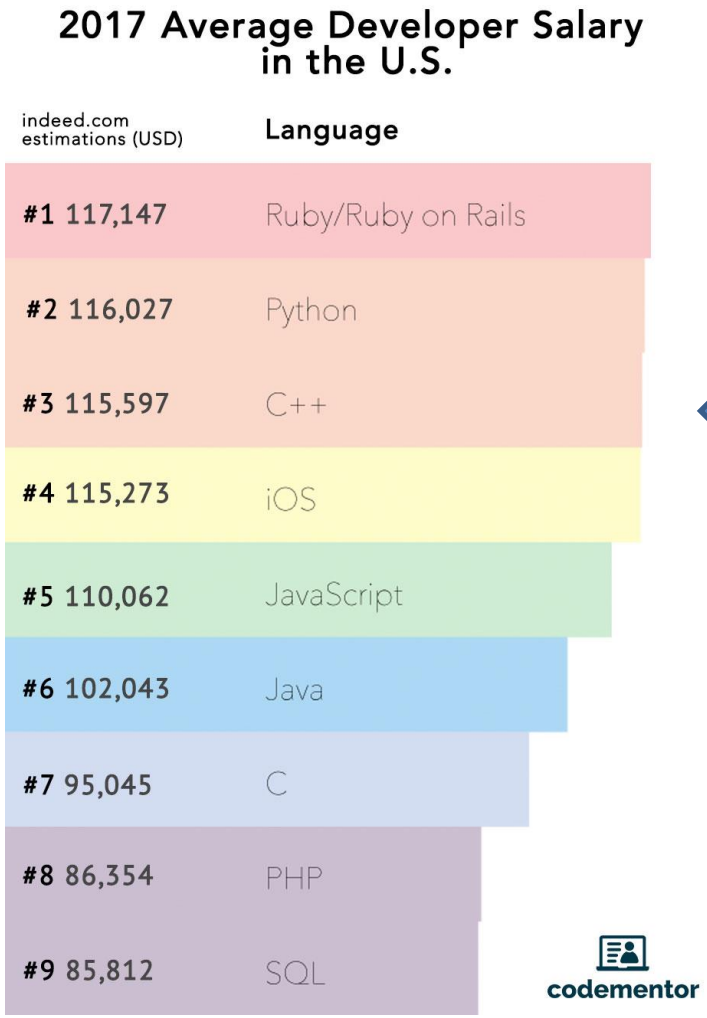


10. 왜 C++인가

C++은 C를 기반으로 한 강력한 언어

- It is designed for programming systems software.
- It has been used to build games/game engines, desktop apps, mobile apps, and web apps.
- Facebook has developed several high performance and high reliability components with it.
- Many softwares have been built with C++, including Adobe Systems, Amazon, PayPal, Chrome, and more.

10. 왜 C++인가

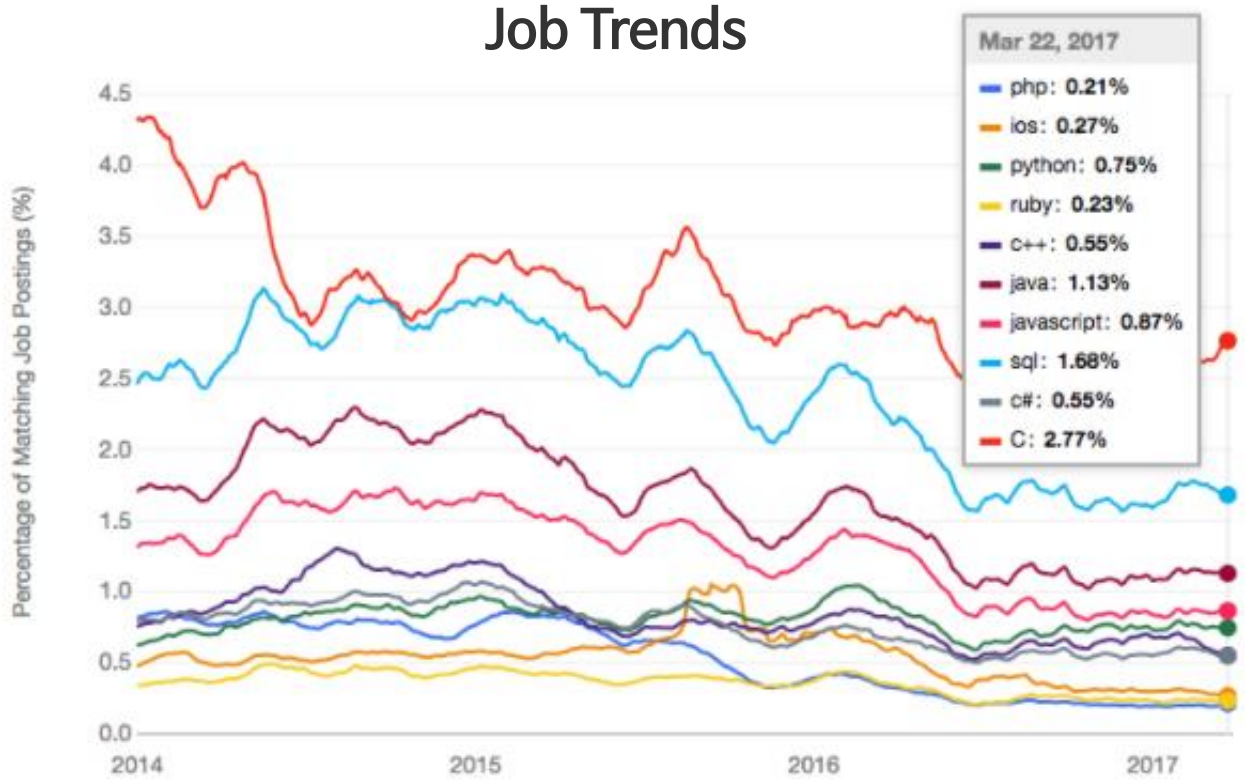


Ranked in Top 3

Reference:

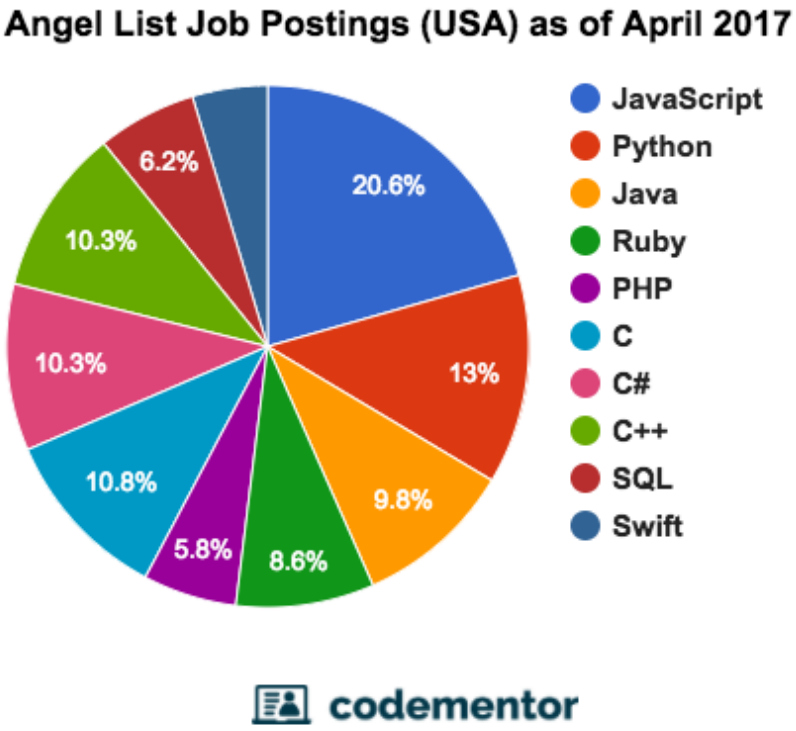
https://www.codementor.io/codementorteam/beginner-programming-language-job-salary-community-7s26wmbm6

10. 왜 C++인가



Rank: **C** / SQL / Java / JavaScript / Python / **C++** …

10. 왜 C++인가



Rank: JavaScript / Python / C / **C++** / Ruby / Java …

강의 계획

주차	내용	
~~1~~	~~객체지향 개요 (lecture note)~~	~~개발 환경 구축~~
~~2~~	~~클래스 (lecture note)~~	~~C++ 기초 문법~~
~~3~~	~~객체 (Chap. 1)~~	~~**프로젝트** 팀 구성~~
4	객체 생성과 활용 (Chap. 2)	
5	추상화 및 정보 은닉 (Chap. 4, Chap. 5)	
6	상속과 구성 (Chap. 14)	**프로젝트** 제안서 제출
7	다형성과 가상함수 (Chap. 15)	
8	**중간고사**	

질문 및 답변