Nested Class

Nested Class

- Nested Class
  - ✓ 별도로 클래스를 정의하기에는 크기가 작은 경우나 하나의 클래스 안에서 구분되어야 하는 정보를 포함해야 하는 경우 클래스 내포에 다른 클래스를 선언해서 사용하는 것이 가능
  - ✓ GUI의 이벤트 처리를 할 때 많이 사용
  - ✓ 종류
    - Instance inner class: instance variable
    - nested (static inner) class: class variable
    - local class : local variable
    - anonymous inner class

Nested Class

- Instance inner class
  - ✓ 클래스의 내포에 클래스를 선언해서 내포 클래스의 인스턴스를 생성해서 사용
  - ✓ 형식

```
class Outer{
        변수
        메서드
    class Inner{
        변수
        메서드
    }
}
```

  - ✓ 클래스 생성 구조
    - Outer.class
    - Outer$Inner.class

Nested Class

- Instance inner class
  - ✓ Outer의 메서드에서 Inner의 변수나 메서드를 호출 시에는 Instance화를 해서 사용
  - ✓ Inner의 메서드에서 Outer의 변수나 메서드를 직접 호출하는 것은 가능
  - ✓ 외부에서 내포 클래스의 Instance화

    바깥클래스 인스턴스명 = new 바깥클래스();

    바깥클래스.안쪽클래스 인스턴스명=바깥클래스의 인스턴스명.new 안쪽 클래스의 생성자();

    Outer ou=new Outer();

    Outer.Inner in=ou.new Inner();

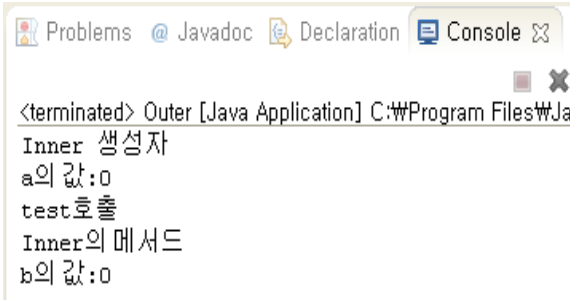
Nested Class

- Instance inner class
  - ✓ 클래스 생성 – Outer.java

```
public class Outer {
    private int a;

    public Outer() {
            System.out.println("Outer의 생성자");
    }

    public void test() {
            System.out.println("test호출");
    }
```



Nested Class

- Instance inner class
  - ✓ 클래스 생성 – Outer.java

```
public class Inner {
        private int b;
        public Inner() {
                System.out.println("Inner 생성자");
                // 사용 가능: Inner클래스 안에서 Outer의 멤버는 바로 사용 가능
                System.out.println("a:" + a);
                test();
        }
        public void method() {
                System.out.println("Inner의 메서드");
                System.out.println("b:" + b);
        }
    }
    public static void main(String args[]) {
        Outer ou = new Outer(); // 결과 : Outer의 생성자
        // ou.method(); //Inner의 메서드를 바로 접근 불가.
        Outer.Inner in = ou.new Inner(); // 결과 : Inner 생성자
        in.method(); // 결과 : Inner의 메서드0
    }
}
```

Nested Class

- static inner class
  - ✓ 내포 클래스를 static 변수처럼 사용
  - ✓ 내포 클래스 안에 static 멤버를 포함시켜야 하거나 외부 클래스의 static 메서드 내에서 내포 클래스를 사용하는 경우에 사용
  - ✓ 일반 내포 클래스에는 static 멤버를 포함시킬 수 없음

```
class Outer{
        변수
        메서드
        static class Inner {
                변수
                메서드
        }
}
```

  - ✓ 클래스의 특성이 static
  - ✓ 외부 클래스의 인스턴스 없이 내포 클래스 인스턴스 생성 가능
  - ✓ 내포 클래스에서는 외부 클래스에 선언된 멤버 중에서 static 멤버 만 사용이 가능

Nested Class

- static inner class
  - ✓ 내포 클래스를 static 변수처럼 사용

```
public class StaticInner {
    int a;
    public static class Inner
    {
            static int g_n;
            static {
                    g_n=1;
            }
            public void disp()
            {
                    System.out.println("g_n 의 값:" +  g_n);
            }
    }
    public static void main(String args[])
    {
            //외부 클래스의 인스턴스 없이도 내포 클래스의 인스턴스 생성 가능
            Inner obj = new Inner();
            obj.disp();
    }
}
```

Nested Class

- Local Inner Class
  - ✓ 클래스를 메서드 내포에 선언하여 지역변수의 형태로 사용

```
class Outer{
        변수
        메서드 {
        class Local   {
                        변수
                        메서드
                       }
    }
}
```

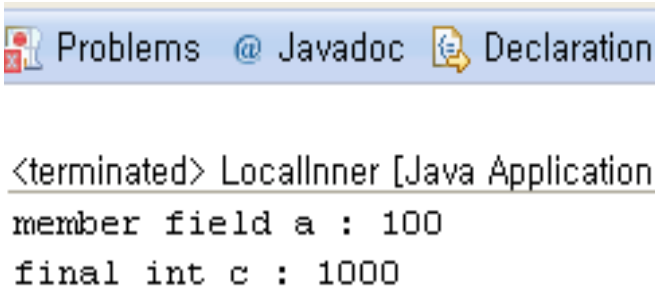
    - Outer.class
    - Outer$1Local.class

  - ✓ Local Inner Class는 포함하고 있는 메서드의 지역 변수를 사용할 수 없으며 final 변수 와 외부 클래스의 멤버 변수나 메서드 사용 가능
  - ✓ 메서드의 호출이 끝나면 자동으로 소멸

Nested Class

- Local Inner Class
  - ✓ LocalInner.java

```
public class LocalInner {
    int a = 100;//멤버 변수
    public void innerTest(int k){
            int b = 200;// 지역변수
            final int c = k; //상수
            class Inner{
                    // Local 내포 클래스는 외부클래스의 멤버 변수와
                    // 상수들만 접근이 가능하다.
                    public void getData(){
                            System.out.println("member field a : "+a);
//                          System.out.println("int b : "+b);
//Local 내포클래스는 지역변수를 사용할 수 없습니다.
                            System.out.println("final int c : "+c);//상수 사용
                    }
            }
            Inner i = new Inner();//메서드내에서 Local 내포 클래스 생성
            i.getData();//생성된 reference를 통해 메서드 호출
    }
```



Nested Class

- Local Inner Class
  - ✓ LocalInner.java

```
        public static void main(String[] args) {
                LocalInner outer = new LocalInner();
                outer.innerTest(1000);//외부 클래스의 멤버 메서드 호출
        }
}
```

Nested Class

- Anonymous Class – 익명 클래스
  - ✓ 클래스를 이름 없이 인스턴스를 바로 생성해서 사용
  - ✓ 인스턴스를 생성하는 시점에 필요한 내용을 완성해서 사용
  - ✓ 특정 순간에만 사용하는 클래스가 필요한 경우에 사용
  - ✓ GUI(윈도우 프로그래밍, 안드로이드) 프로그래밍의 이벤트 Listener 들을 이 방법을 이용해서 사용하는 경우가 많음
  - ✓ 인터페이스를 구현하는 경우 메서드가 1개이면 람다 사용 가능 – Android Studio 에서는 자동 변경

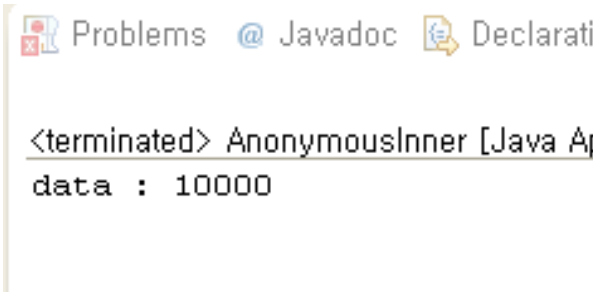
```
class Outer{
        변수
        메서드 {
                new 클래스(){
                        메서드
                        { }
                }.메서드;
}
```

  - Outer.class
  - Outer$1.class

Nested Class

- Anonymous Class – 익명 클래스
  - ✓ AnonymousInner.java

```
public class  AnonymousInner{
    interface TestInter{
            int data = 10000;
            public void printData();
    }
    public void test(){
            new TestInter(){
                    public void printData()//미완성된 것을 완성하여
                    {                                    //JVM이 확인가능하도록 해준
    다.
                                    System.out.println("data : "+data);
                    }
            }.printData();
    }
    public static void main(String[] args){
            new AnonymousInner().test();
    }
}
```



Nested Class

- 내포 인터페이스
  - ✓ 클래스 내포에 인터페이스 선언 가능
  - ✓ 클래스 내포에 인터페이스가 선언되면 static으로 간주해서 외부 인스턴스 생성 없이 사용 가능 – 안드로이드의 이벤트 처리 인터페이스가 이 구조로 많이 만들어져 있음

인스턴스 간의 데이터 공유

데이터 공유

- 데이터 공유
  - ✓ 동일한 클래스로부터 생성된 인스턴스는 클래스에 static 필드를 만들어서 사용
  - ✓ 하나의 인스턴스 안에서 다른 클래스의 인스턴스가 생성되는 경우에 안에서 만들어지는 인스턴스가 외부 인스턴스의 데이터를 사용하고자 하는 경우에는 내포 인스턴스를 생성할 때 생성자를 이용해서 데이터를 넘겨주거나 아니면 setter를 이용해서 데이터를 설정
  - ✓ 내포 인스턴스가 외부 인스턴스에게 데이터를 넘겨주고자 하는 경우에는 내포 인스턴스를 만들 때 외부 인스턴스의 참조를 생성자나 setter를 이용해서 넘겨서 조작할 수 있으면 메서드의 리턴 값 등으로 결과를 받아서 사용
  - ✓ 동일한 클래스가 아니고 서로 아무런 관계가 없는 클래스 인 경우에는 싱글톤 클래스를 만들어서 데이터를 공유 – 하나의 public 클래스를 만들고 public static 변수를 만들어서 사용하는 방법으로 데이터 공유가 가능하지만 권장하지 않음

데이터 공유

- 동일한 클래스로부터 만들어진 데이터 공유
  - ✓ StaticShare.java

```
public class StaticShare {
    //클래스로부터 만들어진 모든 인스턴스가 공유
    public static String share;
    //클래스로부터 만들어진 인스턴스가 각각 소유
    public int each;
}
```

데이터 공유

- 동일한 클래스로부터 만들어진 데이터 공유
  - ✓ Main.java

```
public class Main {
    public static void main(String[] args) {
            StaticShare obj1 = new StaticShare();
            StaticShare obj2 = new StaticShare();
            obj1.share = "안녕하세요";
            obj1.each = 100;

            //share는 static 이므로 obj1이 변경한 내용이 출력
            System.out.println(obj2.share);
            //each는 static이 아니므로 자신의 값 출력
            System.out.println(obj2.each);
    }
}
```

데이터 공유

- Singleton Pattern
  - ✓ 하나의 인스턴스만 생성할 수 있는 디자인 패턴
  - ✓ 시스템 전체에 공통적으로 적용되는 설정 정보를 보관하는 관리자 클래스나 전역 변수를 소유한 클래스를 만들 때 주로 이용
  - ✓ 적용 방법
    - 생성자를 private으로 만들어서 외부에서 인스턴스 생성할 수 없도록 함
    - 자신의 타입으로 된 static 변수를 선언
    - static 변수가 null이면 생성해서 리턴하고 그렇지 않으면 그냥 리턴하는 static 메서드를 생성
  - ✓ jdk의 클래스 중에서 인스턴스 생성 시 생성자를 호출하지 않고 static 메서드를 이용해서 인스턴스를 리턴받는 클래스는 singleton 패턴이 적용된 클래스 인 경우가 있음

데이터 공유

- Singleton Pattern
  - ✓ Singleton.java

```
class OnlyOne {
    static OnlyOne obj = null;
    private OnlyOne(){
    }
    public static OnlyOne getInstance(){
            if(obj==null)
                    obj = new OnlyOne();
            return obj;
    }
}

public class Singleton{
    public static void main(String[] args) {
            OnlyOne obj1 = OnlyOne.getInstance();
            OnlyOne obj2 = OnlyOne.getInstance();

            System.out.println("obj1의 해시코드:" + obj1.hashCode());
            System.out.println("obj2의 해시코드:" + obj2.hashCode());
    }
}
```

데이터 공유

- 포함 관계
  - ✓ EmbedClass.java

```
class EmbedClass{
    private ContainClass parent;
    private int y;

    EmbedClass(ContainClass parent){
            this.parent = parent;
    }

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

    void disp() {
            System.out.println("부모의 x 사용하기:" + parent.x);
            parent.x = 20;
            System.out.println("y:" + y);
    }
}
```

데이터 공유

- 포함 관계
  - ✓ ContainClass.java

```
class ContainClass{
    public int x;
    public EmbedClass embed;
    ContainClass(){
            x = 10;
    }

    void createEmbed() {
            //생성자를 이용한 전달
            embed = new EmbedClass(this);
            //setter를 이용한 전달
            embed.setY(x);
    }

    void print() {
            System.out.println("나의 x 사용하기:" + x);
    }

    void method() {
            embed.disp();
    }
}
```

데이터 공유

- 포함 관계
  - ✓ ContainClass.java

```
public class IncludeMain {

    public static void main(String[] args) {
            ContainClass obj = new ContainClass();
            obj.print();
            obj.createEmbed();
            obj.print();
            obj.method();
    }
}
```

Java Doc

- Javadoc
  - ✓ 애플리케이션 및 라이브러리 개발에는 여러 프로그래머가 참여하게 되는데 그 때 클래스나 메서드에 대한 설명을 만들지 않으면 다른 프로그래머가 만든 것을 이해하기 어려움
  - ✓ Javadoc은 자바의 클래스나 메서드에 기술한 주석으로부터 HTML 형식의 API 문서를 생성하는 도구
  - ✓ Javadoc이 해석할 수 있도록 소스 코드에는 Javadoc 형식의 문서화 주석을 기술
  - ✓ 한 줄 주석은 '//'으로 시작하고 여러 줄 주석은 '/*'로 시작하는 반면, Javadoc 주석은 '/**'로 시작
  - ✓ 태그(@param 등)를 이용해서 메타 정보를 기술
  - ✓ Javadoc 주석을 쓰면 다음과 같은 이점
    - 이클립스 등의 IDE에서 메서드에 마우스 커서를 위치시키면 툴팁으로 메서드의 Javadoc이 표시되기 때문에 개발 효율이 향상
  - ✓ Javadoc 주석을 기술하는 부분과 내용
    - 클래스: 클래스가 어떤 역할을 갖는지(데이터를 갖고 있을 뿐인지, 통신을 하는지, 어떤 처리 로직을 갖고있는지)를 기술하고 버전이나 작성자 등의 정보를 기술
    - 필드: 해당 필드가 클래스 내에서 어떠한 의미가 있는 데이터를 보관하고 있는지를 기술
    - 메서드: 메서드의 입출력을 기술하고 어떤 인수에 대해 어떤 반환값을 반환하는지, 또는 어떤 외부 처리를 할 것인지를 기술하며 발생하는 예외에 대한 설명도 기술

Java Doc

- Javadoc
  - ✓ 클래스 주석

```
/**
 * 사원 정보의 등록/변경/삭제를 실시하는 업무 로직 클래스
 */
public class EmployeeService {
```

  - ✓ 필드 주석

```
/** 사원 정보에 액세스하기 위한 Dao */
private EmployeeDao employeeDao;
```

  - ✓ 메서드 주석

```
/**
 * 사원 정보를 업데이트한다
 */
public int updateEmployee(int employeeId, String employeeName) throws
    SQLException {
```

Java Doc

- Javadoc 주석의 태그
  - ✓ @author: 클래스나 메서드의 작성자를 기술하며 여러 @author를 기술함으로써 여러 작성자를 나타낼 수도 있음
  - ✓ @deprecated: 폐지되거나 폐지될 가능성이 있는 클래스나 필드, 메서드
  - ✓ @param: 변수명 설명으로 메서드의 인수 및 제네릭 타입에 대한 설명을 기술
  - ✓ @return: 메서드의 반환값에 대한 설명을 기술
  - ✓ @since: 버전 클래스나 메서드가 생성된 버전을 기술
  - ✓ @throws: 예외 클래스 설명 메서드에서 throw되는 예외에 대한 설명을 기술
  - ✓ @version: 버전 클래스나 메서드의 현재 버전을 기술

Java Doc

- Javadoc 주석의 태그
  - ✓ 클래스 주석

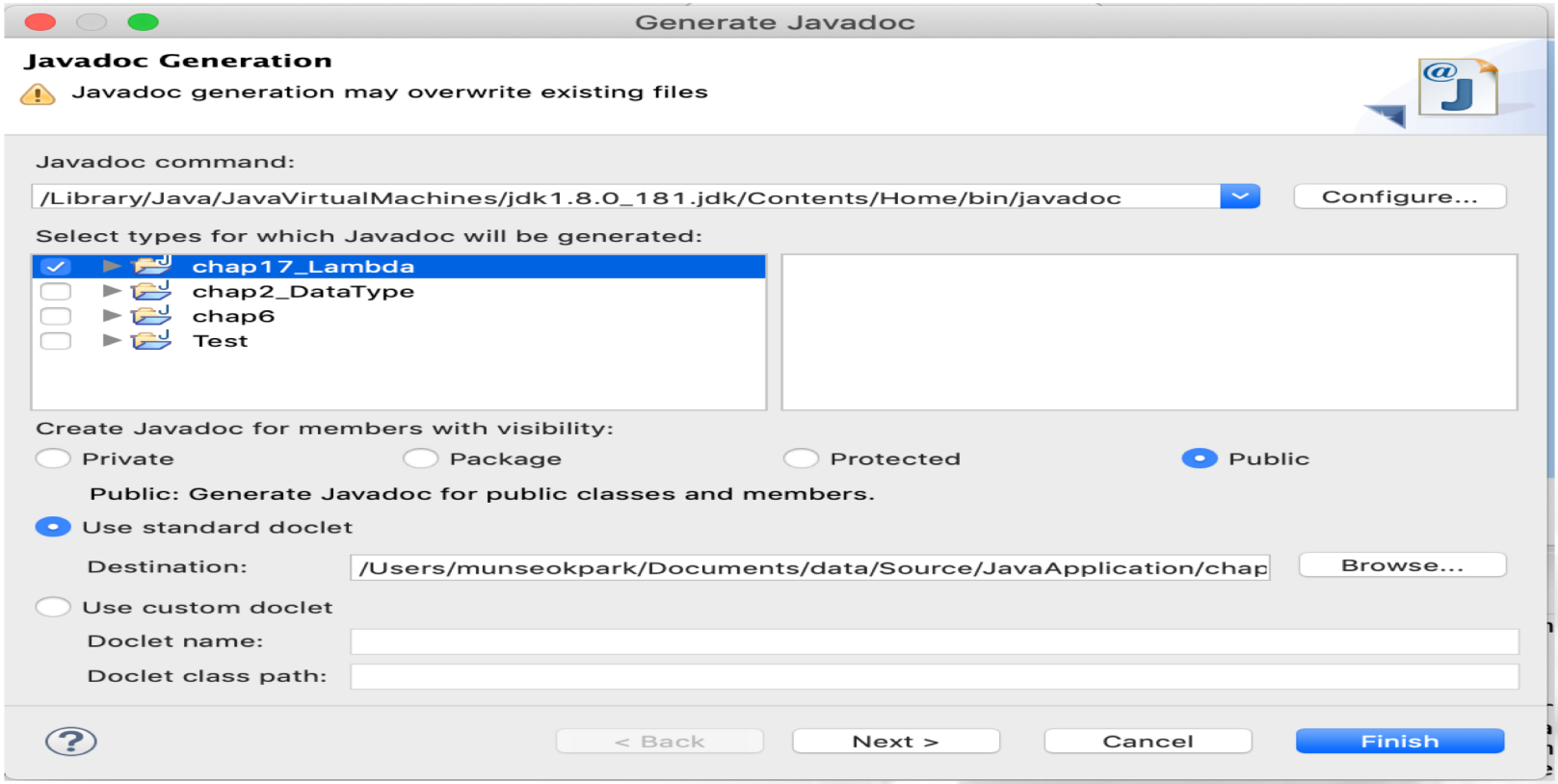
```
/**
 * 사원 정보의 등록/변경/삭제를 실시하는 업무 로직 클래스
 * @author S. Tanimoto
 * @version 2.1
 * @since 1.0
 */
public class EmployeeService {
```

  - ✓ 메서드 주석

```
/**
 * 사원 정보를 업데이트한다
 * @param employeeId 업데이트 대상의 사원ID
 * @param employeeName 업데이트 후의 사원 이름
 * @return 업데이트한 건 수
 * @throws SQLException 업데이트 시에 오류가 발생한 경우
 */
public int updateEmployee(int employeeId, String employeeName) throws
    SQLException {
```

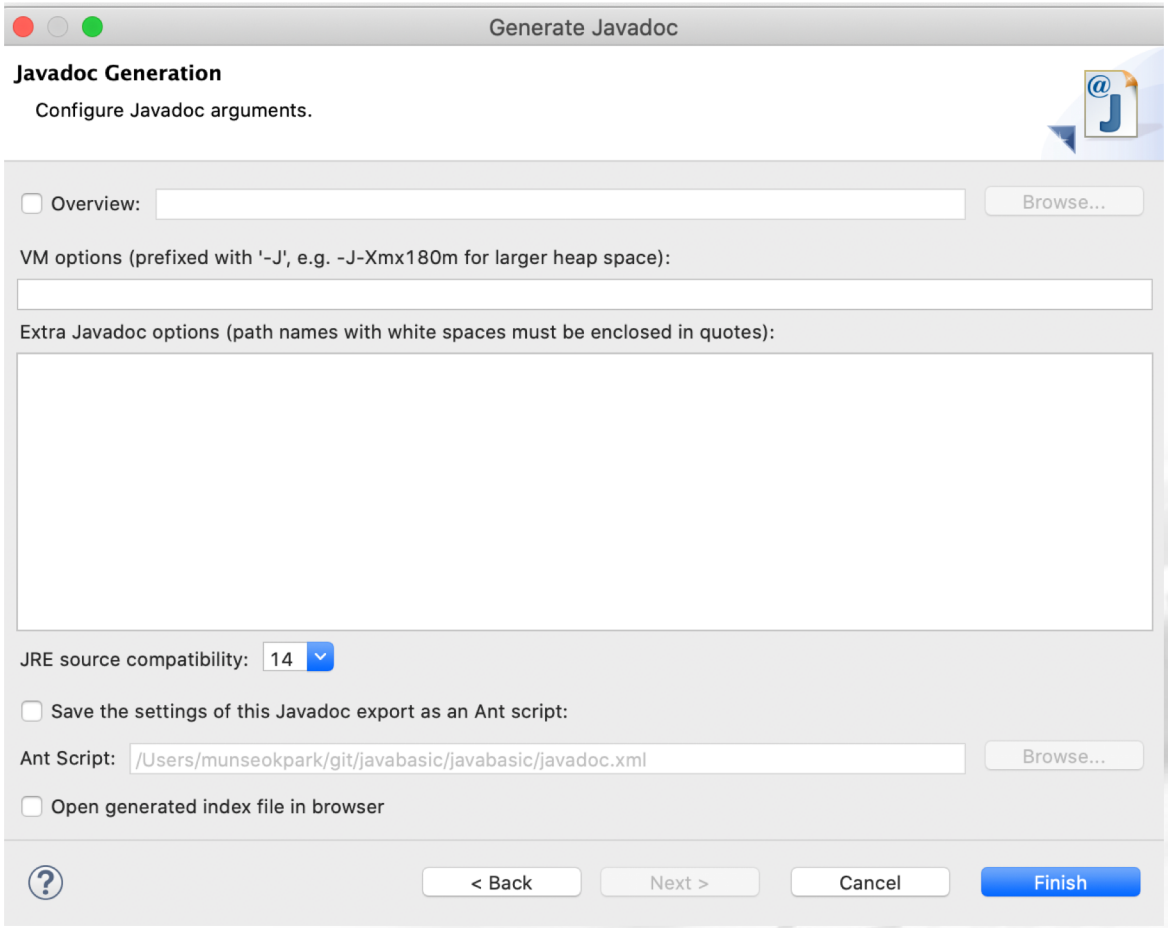
Java Doc

❖ [Project] – [Generate Javadoc]를 클릭



Java Doc

❖ Windows에서 인코딩 문제로 문서가 만들어지지 않는 경우 VM options 란에 추가
-locale ko_KR -encoding UTF-8 -charset UTF-8 -docencoding UTF-8



Java Doc

- HTML 태그 사용
  - ✓ Javadoc 주석은 IDE 및 도구에 의해 자동으로 포맷되어 표시되지만 임의의 포맷으로 실시하려면 HTML 태그를 사용하여 의도적으로 줄 바꿈을 하거나 들여쓰기를 지정
  - ✓ 표 등의 복잡한 태그를 사용할 수도 있지만 많이 사용하면 자동 포맷된 표시가 망가져 버릴 가능성도 있기 때문에 HTML 태그의 사용은 최소로 하는 것이 좋음
  - ✓ < 나 > 등의 기호를 그대로 문자로 사용할 수 있는데 이 경우 {@literal} 또는 {@code}를 사용하여 기술하면 '<'와 '>'라고 기술하지 않아도 되는데 {@literal}은 보통의 문장에 사용하고 {@code}는 코드를 기술할 때 사용

Java Doc

- Javadoc 주석 안에서 다른 클래스나 메서드를 참조하고 싶은 경우가 있는데 그때는 {@link} 태그를 사용
  - ✓ 클래스를 참조하는 경우

    {@link 클래스명}

    {@link EmployeeService}
  - ✓ 자신의 클래스의 메서드를 참조하는 경우(Javadoc을 기술하는 클래스 내에 참조하려는 메서드가 있는 경우)

    {@link #메서드명(인수의 타입, 인수의 타입, ...)}

    {@link #updateEmployee(int, String)}
  - ✓ 다른 클래스의 메서드를 참조하는 경우

    {@link 클래스명 #메서드명(인수의 타입, 인수의 타입, ...)}

    {@link EmployeeService#update(Employee)}

Java Doc

- 상속
  - ✓ 클래스를 상속하고 메서드를 오버라이드하는 경우 슈퍼 클래스의 설명과 같은 문장을 쓰는 것은 시간을 낭비하는 일이므로 슈퍼 클래스의 Javadoc 주석의 설명을 그대로 사용하는 것이 효율적
  - ✓ {@inheritDoc}을 이용하면 슈퍼 클래스의 Javadoc 주석의 글을 참조
  - ✓ 문장을 참조시키는 범위는 {@inheritDoc}을 기술하는 부분에 따라 인수 하나부터 메서드 전체까지 다양
  - ✓ @return {@inheritDoc} : @return 주석 한 개만 가져오기
  - ✓ {@inheritDoc}: 상위 클래스의 주석 전체 가져오기

UML

UML

UML

- 분석, 설계를 비주얼 화, 문서화 하기 위한 그래픽 언어
- Unified
  - 이전의 OO 방법들의 통합
- Modeling
  - 객체지향 분석 설계를 위한 비주얼 모델링
- Language
  - 모형화된 지식(의미)을 표현

UML

- UML
  - ✓ 시스템에 대한 지식을 찾고 표현하기 위한 언어
  - ✓ 시스템을 개발하기 위한 탐구 도구
  - ✓ 비주얼 모델링 도구
  - ✓ 근거가 잘 정리된 가이드 라인
  - ✓ 분석, 설계 작업의 마일스톤
  - ✓ 실용적 표준
  - ✓ 비주얼 프로그래밍 언어나 데이터베이스 표현도구 아님
  - ✓ 개발 프로세스나 품질 보증 방안이 아님
  - ✓ 시스템의 구조를 나타내는 6개의 구조 다이어그램과 시스템 동작을 표현하는 7개의 행위 다이어그램으로 표현
  - ✓ 구성 요소는 사물, 관계, 다이어그램

UML

- UML
  - ✓ 사물(Things)
    - 모델을 구성하는 가장 중요한 기본 요소로, 다이어그램 안에서 관계가 형성될 수 있는 대상들
    - 구조(Structural) 사물: 시스템의 개념적, 물리적 요소를 표현
    - 행동(Behavioral) 사물: 시간과 공간에 따른 요소들의 행위를 표현
    - 그룹(Grouping) 사물: 요소들을 그룹으로 묶어서 표현
    - 주석(Annotation) 사물: 부가적인 설명이나 제약조건 등을 표현
  - ✓ 관계(Relationship)
    - 사물과 사물 사이의 연관성을 표현하는 것
    - 연관(Association) 관계: 2개 이상의 사물이 서로 관련되어 있음을 표현함
    - 집합(Aggregation) 관계: 하나의 사물이 다른 사물에 포함되어 있는 관계를 표현함
    - 포함(Composition) 관계: 집합 관계의 특수 한 형태로 포함하는 사물의 변화가 포함되는 사물에게 영향을 미치는 관계를 표현함
    - 일반화(Generalization) 관계: 하나의 사물이 다른 사물에 비해 더 일반적인지 구체적인지를 표현함
    - 의존(Dependency) 관계: 연관 관계와 같이 사물 사이에 서로 연관은 있으나 필요에 의해 서로에게 영향을 주는 짧은 시간 동안 만 연관을 유지하는 관계를 표현함
    - 실체화(Realization) 관계: 사물이 할 수 있거나 해야 하는 기능(행위, 인터페이스)으로 서로를 그룹화 할 수 있는 관계를 표현함

UML

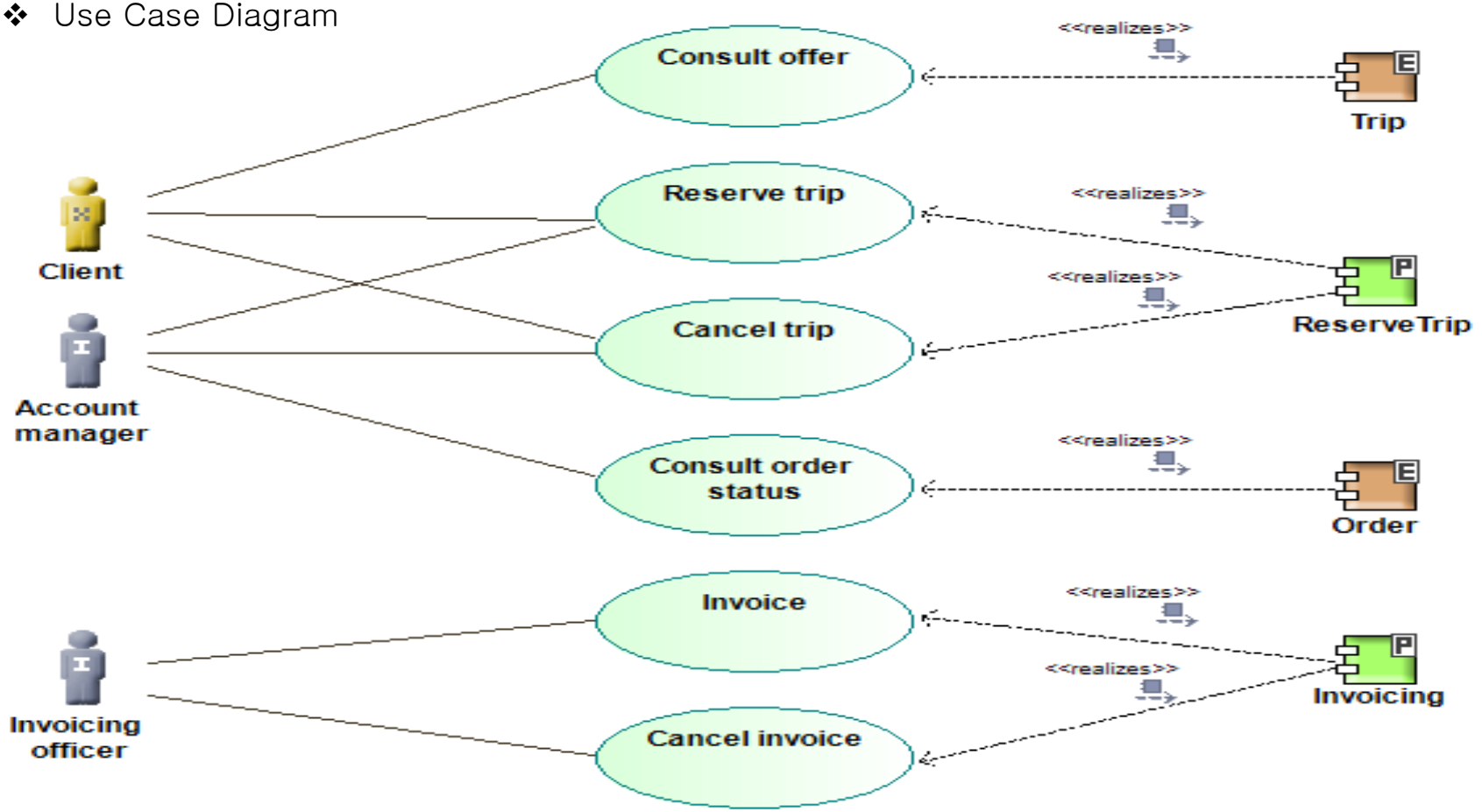
- Diagram
  - ✓ 사물과 관계를 도형으로 표현한 것
  - ✓ 정적 모델링에서는 주로 구조적 다이어그램을 사용하고 동적 모델링에서는 주로 행위 다이어그램을 사용
    - 정적 모델링: 사용자가 요구한 기능을 구현하는데 필요한 자료들의 논리적인 구조를 표현한 것
    - 동적 모델링: 시스템의 내포 구성 요소들의 상태가 시간의 흐름에 따라 변화하는 과정과 변화하는 과정에서 발생하는 상호 작용을 표현한 것
  - ✓ 구조적 다이어그램: Class Diagram, Object Diagram, Component Diagram, Deployment Diagram, Composite Structure Diagram, Package Diagram
  - ✓ 행위 다이어그램: Use Case Diagram, Sequence Diagram, Communication Diagram, State Diagram, Activity Diagram, Interaction Overview Diagram, Timing Diagram

UML

- Use Case Diagram
  - ✓ 기능 모델링을 위한 도구
  - ✓ 개발될 시스템과 관련된 외부 요소들, 즉 사용자와 다른 외부 시스템들이 개발될 시스템을 이용해 수행할 수 있는 기능을 사용자의 관점(View)에서 표현한 것으로 기능 모델링을 하기 위한 도구 중 하나
  - ✓ 구성 요소: 시스템 범위, 액터, 유스케이스, 관계
    - 시스템 범위 (System Scope): 시스템 내포에서 수행되는 기능들을 외부 시스템과 구분하기 위해 시스템 내포의 유스케이스들을 사각형으로 묶어 시스템의 범위를 표현
    - 액터(Actor): 시스템과 상호작용을 하는 모든 외부 요소로 사람이나 외부 시스템을 의미
    - 유스케이스(Use Case): 사용자가 보는 관점에서 시스템이 액터에게 제공하는 서비스 또는 기능을 표현한 것
    - 관계(Relationship): 유스케이스 다이이그램에서 관계는 액터와 유스케이스, 유스케이스와 유스케이스 사이 에서 나타날 수 있으며 포함 관계, 확장 관계, 일반화 관계의 3종류가 있음

UML

- Use Case Diagram

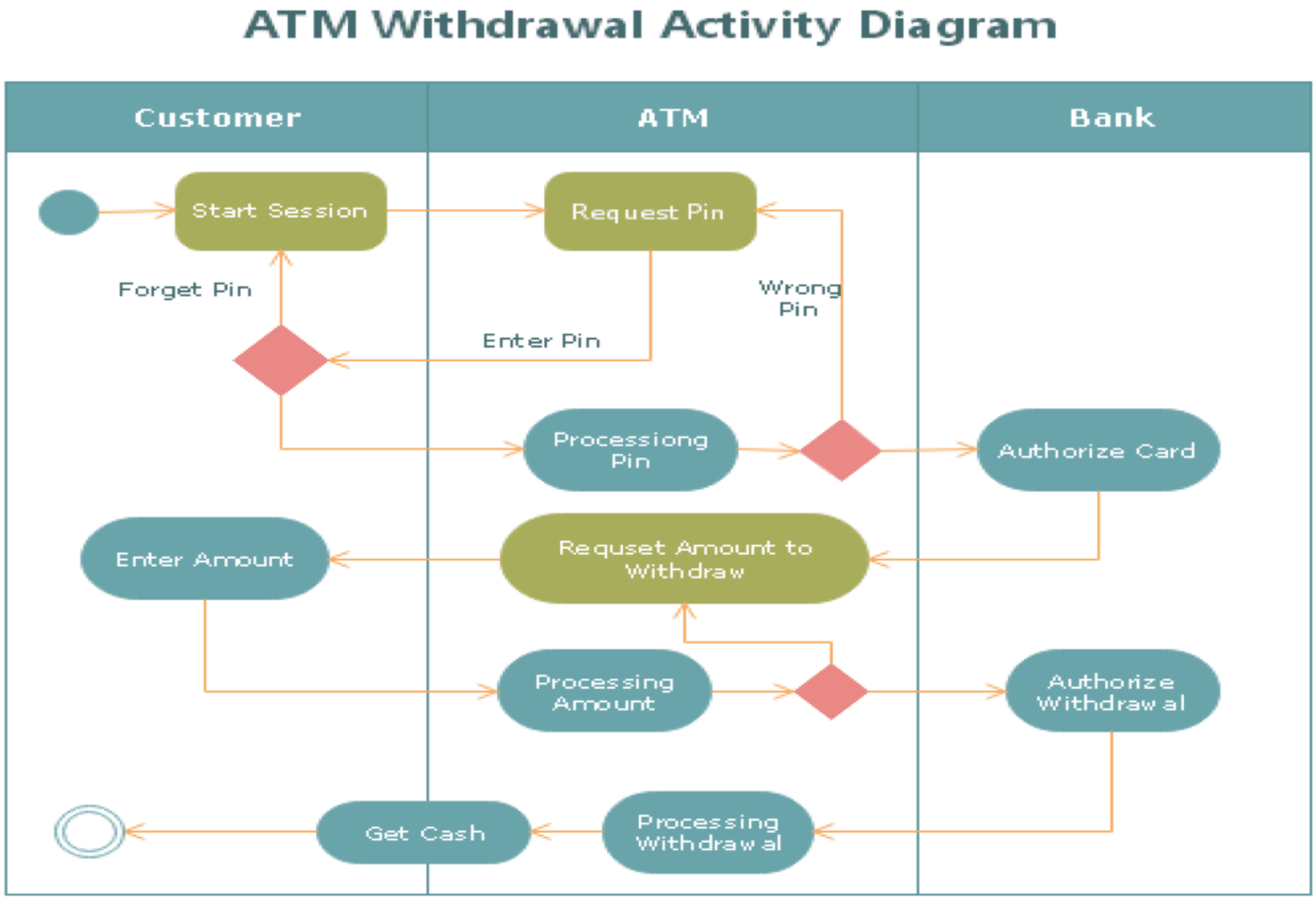


UML

- Activity Diagram
  - ✓ 기능 모델링을 위한 도구
  - ✓ 자료 흐름도와 유사한 것으로 사용자의 관점(View)에서 시스템이 수행하는 기능을 처리 흐름에 따라 순서대로 표현한 것
  - ✓ 활동 다이어그램의 구성 요소 : 액션, 액티비티, 노드, 스윔 레인 등
    - 액션(Action): 더 이상 분해할 수 없는 단일 작업
    - 액티비티(Activity): 몇 개의 액션으로 분리될 수 있는 작업
    - 노드(Node)
      - 시작 노드: 액션이나 액티비티가 시작 됨을 의미
      - 종료 노드: 액티비티 안의 모든 흐름이 종료됨을 의미
      - 조건(판단) 노드: 조건에 따라 제어의 흐름이 분리됨을 표현
      - 병합 노드 : 여러 경로의 흐름이 하나로 합쳐짐을 표현
      - 포크(Fork) 노드: 액티비티의 흐름이 분리되어 수행됨을 표현
      - 조인(Join) 노드: 분리되어 수행되던 액티비티의 흐름이 다시 합쳐 짐을 표현
    - 스윔 레인(Swim Lane): 액티비티 수행을 담당하는 주체를 구분

UML

- Activity Diagram

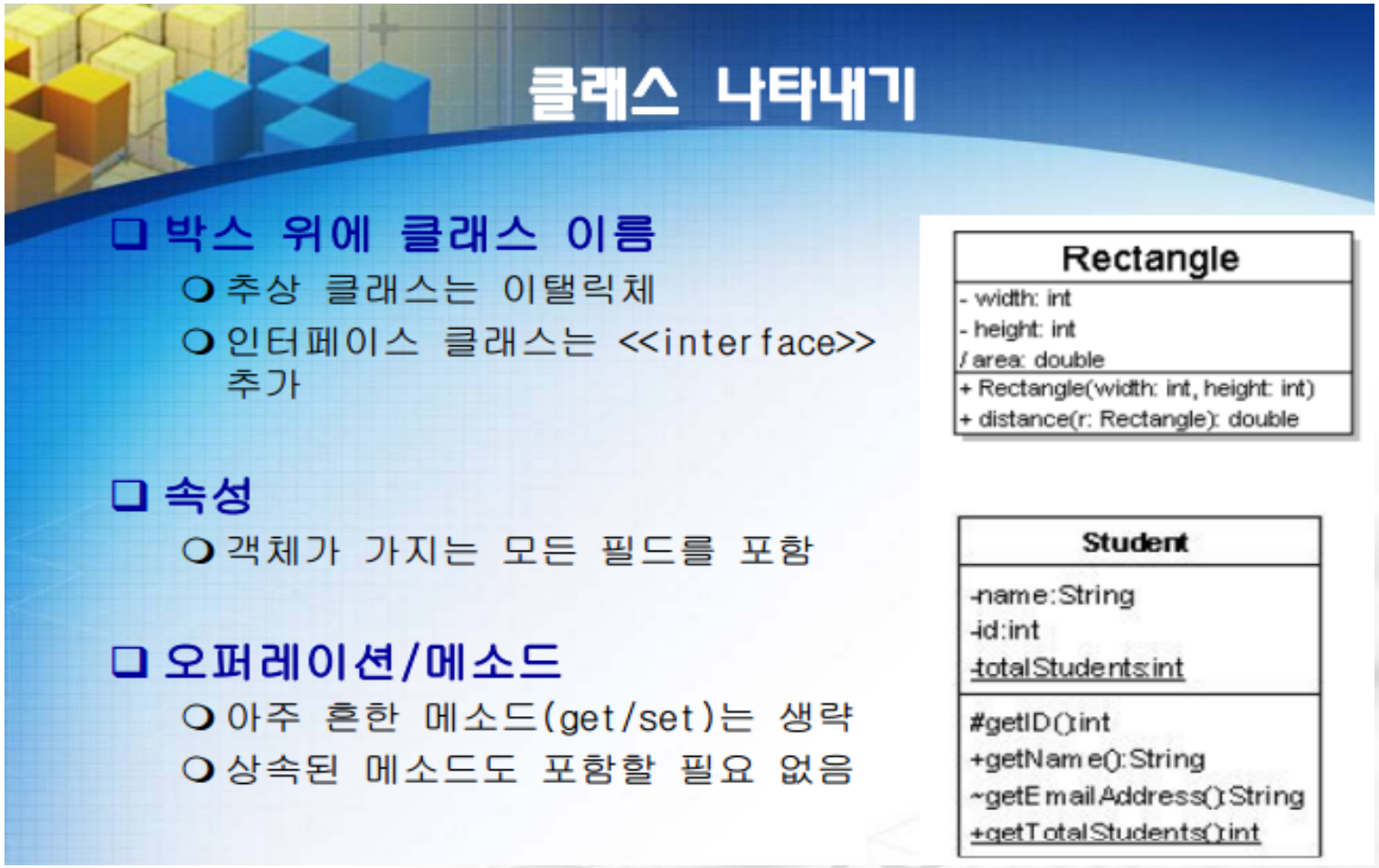


UML

- Class Diagram
  - ✓ 클래스가 가지는 속성 및 메서드 그리고 클래스 사이의 관계를 표현
  - ✓ UML을 이용한 정적 모델링의 대표적인 것이 클래스 다이어그램
  - ✓ 클래스 다이어그램의 구성 요소
    - 클래스(Class): 각각의 객체들이 갖는 속성과 오퍼레이션 (동작)을 표현
      - 일반적으로 3개의 구획(Compartment)으로 나누어서 클래스의 이름, 속성, 오퍼레이션을 표기
      - 속성(Attribute): 클래스의 상태나 정보를 표현
      - 오퍼레이션(Operation, 연산): 클래스가 수 행할 수 있는 동작으로, 함수(메서드, Method)라고도 함
    - 제약조건(Constraint): 속성에 입력된 값에 대한 제약조건이나 오퍼레이션 수행 전 후에 지정해야 할 조건
    - 관계(RelationShip)
      - Association(연관): 하나의 클래스가 다른 클래스 안에서 사용되는 경우
      - Aggregation(집합): 다른 클래스에 존재하는 메서드의 매개변수로만 사용되는 경우
      - Composition(포함): 두 개의 클래스가 서로 다른 클래스를 사용하는 경우
      - Generalization(일반화): 상속 관계
      - Dependency(의존): 특정 메서드를 호출할 때만 영향을 받는 클래스와의 관계

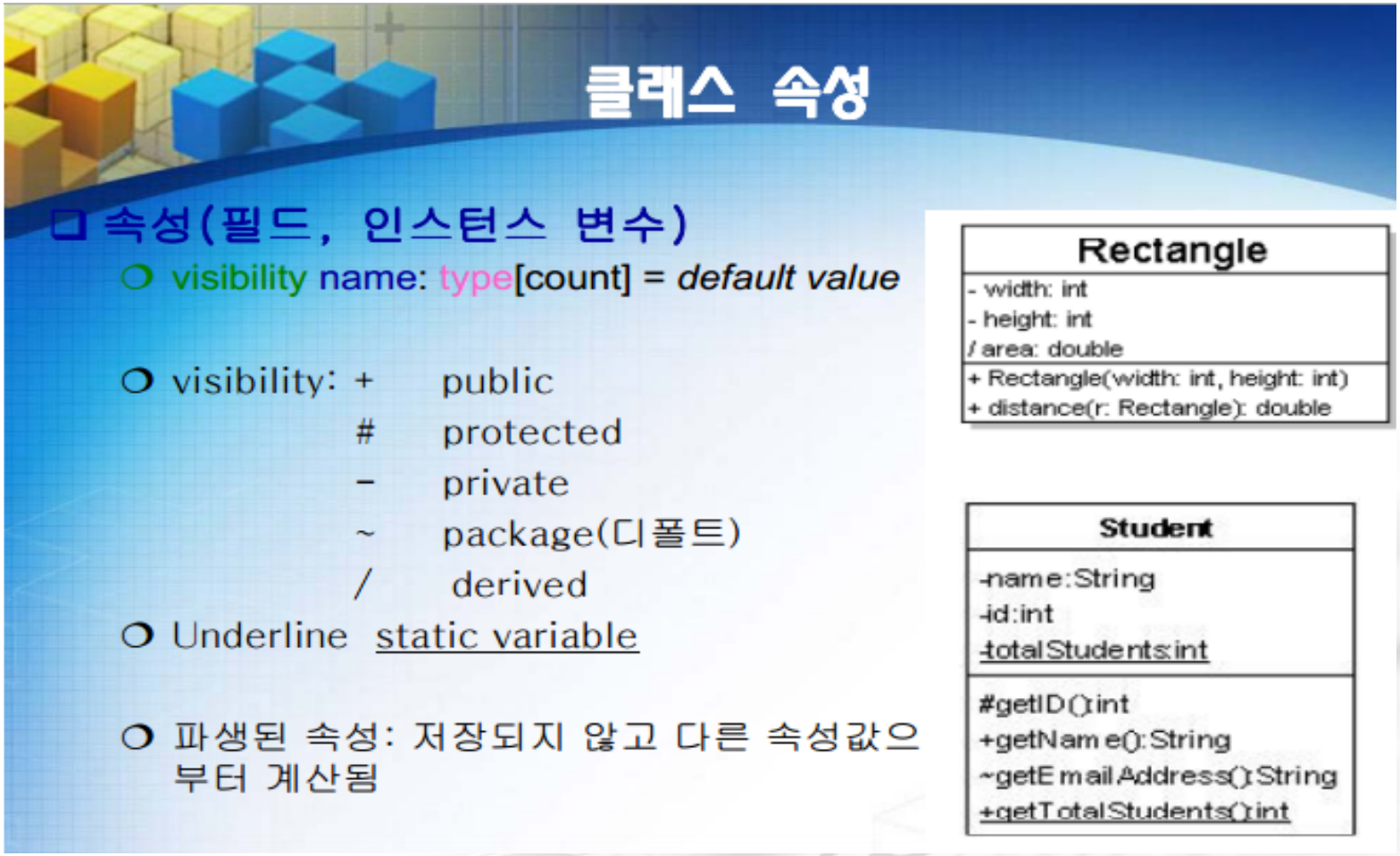
UML

❖ Class Diagram



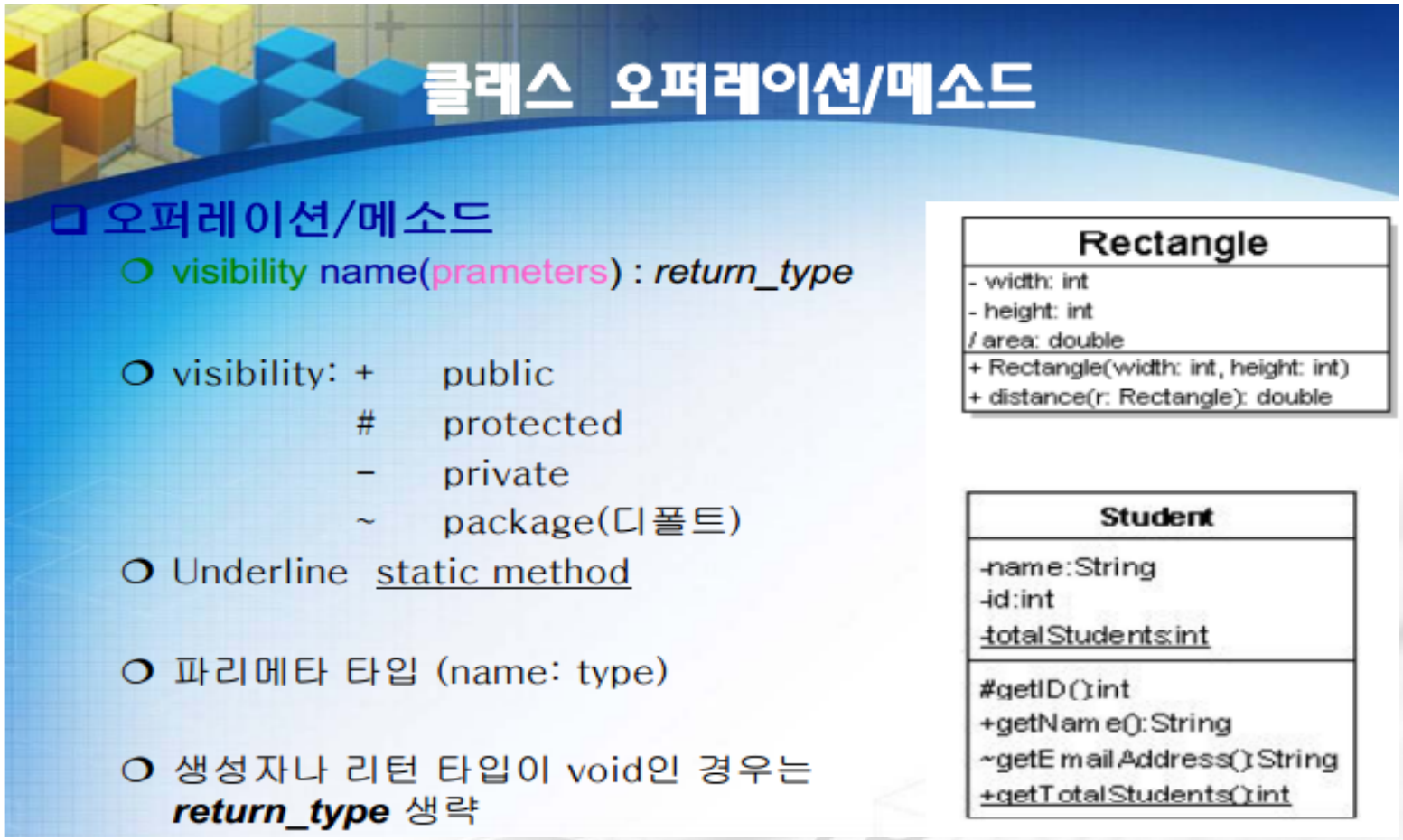
Class Diagram

❖ Class Diagram



Class Diagram

❖ Class Diagram



Class Diagram

❖ Class Diagram

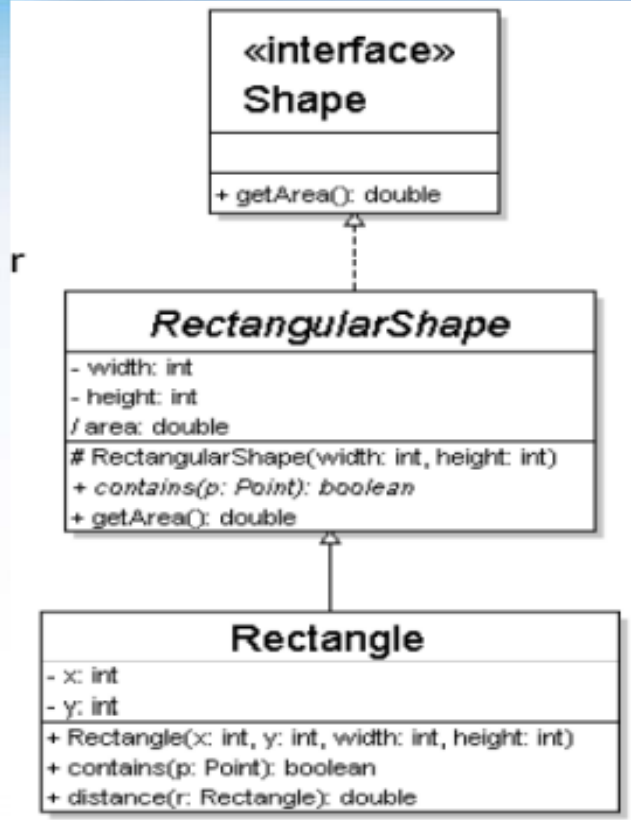
Class Diagram

❖ Class Diagram

일반화 관계

❑ 일반화(상속)

- 부모를 향한 화살표로 표시되는 하향 계층 관계
- 선/화살표는 부모 클래스의 종류에 따라 다름
  - ❖ 클래스:
    실선/검은 헤드 화살표
  - ❖ 추상 클래스:
    실선/흰 헤드 화살표
  - ❖ 인터페이스:
    점선/흰 헤드 화살표



Class Diagram

❖ Class Diagram

Class Diagram

❖ Class Diagram