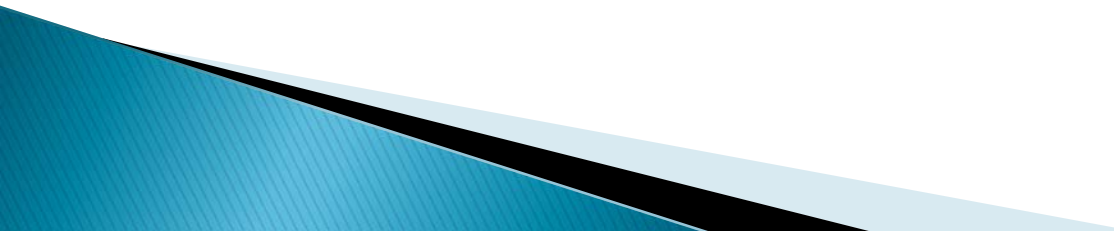


Java Web Programming 입문 08

(Java #08)

오늘의 키워드

- ▶ 다형성 (Polymorphism)
 - 추상 클래스 (Abstract Class)
 - 인터페이스 (Interface)
 - ▶ 예외처리 (Exception Handling)
 - ▶ 컬렉션 (Collection)
 - ▶ 그 외 다루지 않은 부분
- 

추상 클래스 (Abstract Class)

▶ Abstract 의 사전적 의미

□ **abstract** [ˈæbstrækt, ˈæbstrækt, æbˈstrækt]

1. [형용사] 추상적인

abstract knowledge/principles ▶ 추상적인 지식/원칙들

The research shows that pre-school children are capable of thinking in abstract terms. ▶ 그 연구는 미취학 아동들이 추상적인 사고를 할 수 있음을 보여준다.

2. [형용사] 관념적인, 추상적인

We may talk of beautiful things but beauty itself is abstract. ▶ 우리가 아름다운 것들에 대해 말할 수는 있지만 아름다움 자체는 관념적인 것이다.

3. [형용사] 예술 작품이 추상적인

4. [명사] 추상화

5. [명사] 개요, 초록(抄錄)

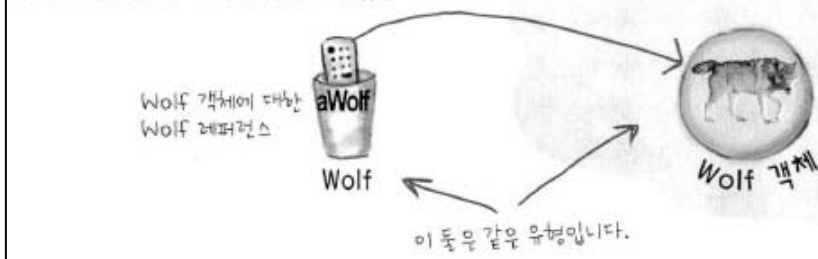
□ **추상¹** (抽象)

1. [명사] <심리> 여러 가지 사물이나 개념에서 공통되는 특성이나 속성 따위를 추출하여 파악하는 작용. 정녕 해가 있다면 그것은 당신들이 지금 알고 있는 것이 아니라 그 이름이 가진 어떤 추상일 뿐이오.

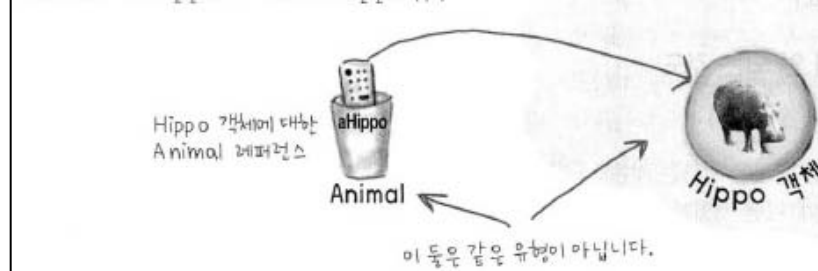
추상 클래스 (Abstract Class)

▶ Animal 클래스 (revisit)

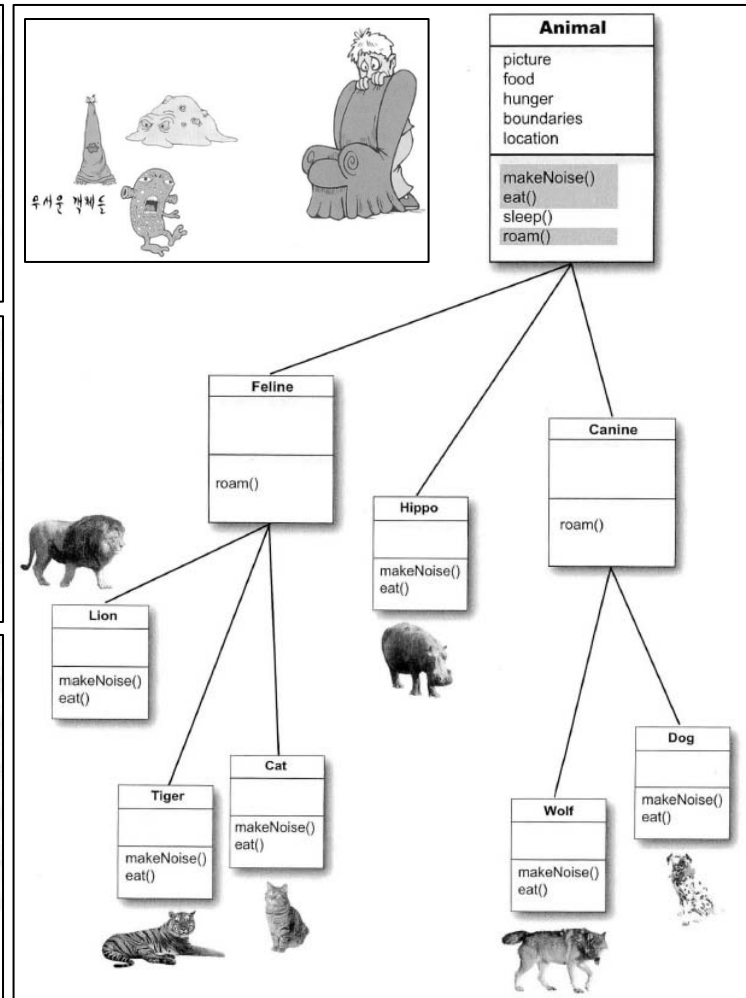
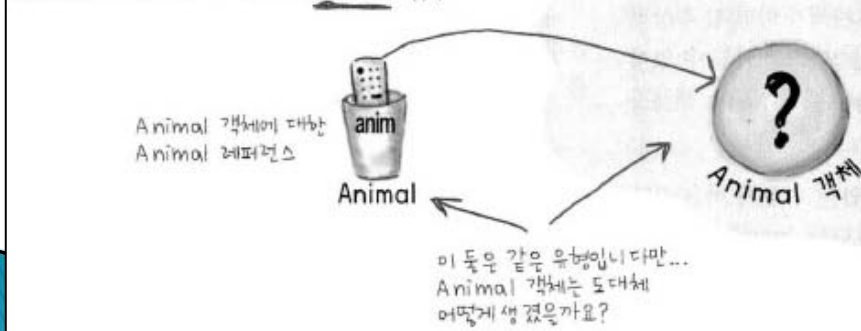
```
Wolf aWolf = new Wolf();
```



```
Animal aHippo = new Hippo();
```



```
Animal anim = new Animal();
```



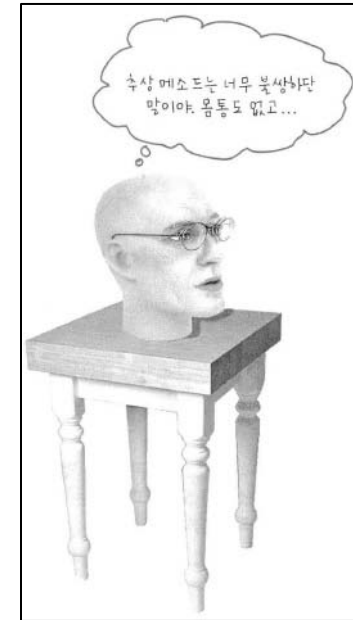
추상 클래스 (Abstract Class)

- ▶ Class : 설계도 (악보)
 - Abstract Class
 - 완성되지 않은 설계도 (완성되지 않은 악보)
- ▶ 미완성 Method를 포함한다는 의미
 - 설계도에 화장실, 부엌, 방, 거실 등은 나누어져 있음
 - 화장실, 부엌은 내부까지 설계가 되어있는데,
 - 무슨 용도로 쓰는 공간인지 나누어져 있기만 한 공간이 있다.
- ▶ **Instance 생성이 불가능**하다. (무조건 상속)
 - Method가 모두 완성되어 있어도
 - (생긴 건 일반 Class와 같아도)
 - Abstract Class로 만들면 Instance 생성이 불가능함
- ▶ 추상 클래스(Abstract Class)

```
abstract class ClassName {  
    // ...  
}
```

- ▶ 추상 메소드(Abstract Method)

```
abstract ReturnType MethodName() ;
```



추상[抽象] - 낱말의 구체적 표상(表象)이나 개념에서 공통된 성질을 뽑아 이를 일반적인 개념으로 파악하는 정신 작용

추상화 - 클래스간의 공통점을 찾아내서 공통의 조상을 만드는 작업
구체화 - 상속을 통해 클래스를 구현, 확장하는 작업

추상 클래스 (Abstract Class)

▶ 추상 클래스, 추상 메소드, 구현

■ 추상 메서드와 추상 클래스의 예

```
1 public abstract class AbstractTest{ //추상 클래스
2     public abstract void abstractMethod(); //추상 메서드
3 }
```

```
1 /**
2  추상 클래스 - 2개의 추상 메서드를 포함한 추상 클래스의 예
3  */
4 public abstract class EmptyCan {
5     public abstract void printContent(); //추상 메서드
6     public abstract void printName(); //추상 메서드
7 }
```

■ 클래스 이름 앞에 abstract 키워드를 사용하지 않았을 때 발생하는 에러

```
1 c:\javasrc\chap05>javac AbstractTest.java
2 AbstractTest.java:2: missing method body, or declare abstract
3     public void abstractMethod();
4             ^
5 1 error
```

■ 추상 클래스의 상속

```
1 public class BeerCan extends EmptyCan{
2     //.....
3 }
```

■ 추상 메서드의 구현

```
1 public class BeerCan extends EmptyCan{ //추상 클래스의 상속
2     public void printContent(){...} //추상 메서드의 구현
3     public void printName(){...} //추상 메서드의 구현
4 }
```

■ 추상 클래스

```
1 public abstract class EmptyCan {
2     //.....
3 }
```

■ 추상 메서드

```
1 public abstract void printContent(); //추상 메서드
2 public abstract void printName(); //추상 메서드
```

```
1 /**
2  추상 클래스의 구현 - EmptyCan을 상속받아 완전한 클래스 만들기
3  */
4 public class BeerCan extends EmptyCan{
5     public void printContent() { //EmptyCan의 printContent() 구현
6         System.out.println("흑맥주");
7     }
8     public void printName() { //EmptyCan의 printName() 구현
9         System.out.println("맥주캔입니다.");
10    }
11    public void sayHello() { //새로운 멤버 메서드 추가
12        System.out.println("안녕하세요 맥주캔입니다.");
13    }
14    public static void main(String args[]) {
15        BeerCan b = new BeerCan();
16        b.printContent();
17        b.printName();
18        b.sayHello();
19    }
20 }
```

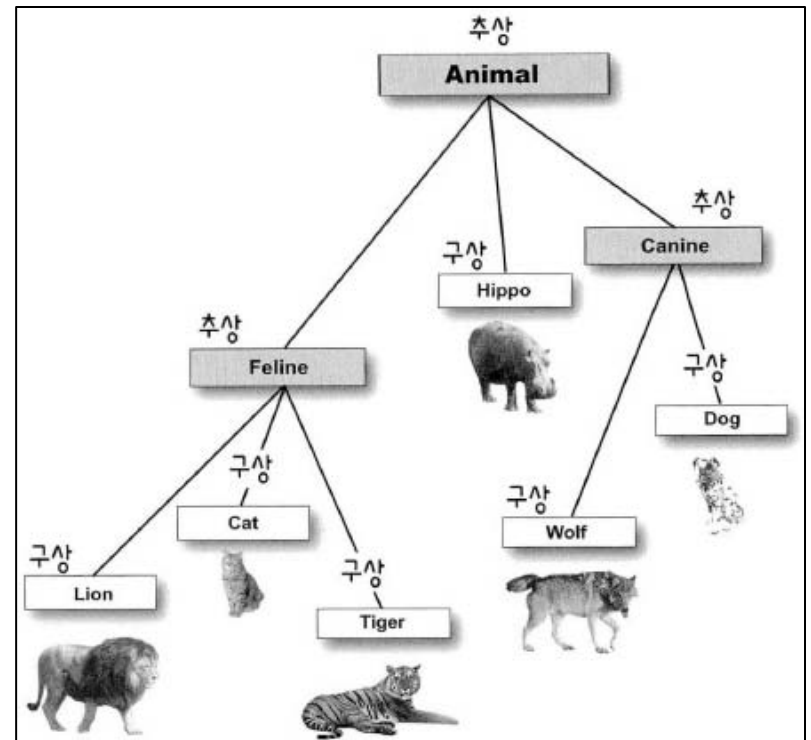
■ 완전한 클래스 BeerCan의 객체 생성 및 메서드 호출

```
1 BeerCan b = new BeerCan();
2 b.printContent();
3 b.printName();
4 b.sayHello();
```


추상 클래스 (Abstract Class)

▶ 구상 클래스 (Concrete Class)

- 추상 클래스가 아닌 클래스
- 인스턴스 생성 가능
- 추상 클래스를 구체화
- 추상 메소드를 구현



추상 클래스 (Abstract Class)

▶ Example

```
abstract class Player {
    boolean pause; // 일시정지 상태
    int currentPos; // 현재 Play 위치

    Player() { // 추상클래스도 생성자를 가짐
        pause = false;
        currentPos = 0;
    }
    // 지정된 위치(pos)에서 재생 시작 기능 작성
    abstract void play (int pos); // 추상 메소드
    // 재생을 즉시 멈추는 기능 작성
    abstract void stop (); // 추상 메소드

    void play() {
        play(currentPos); // 추상 메소드 사용
    }

    void pause() {
        if (pause) { // pause가 true일때(정지)
            pause = false; // false로 바꾼후 현재 위치에서 play
            play(currentPos);
        } else { // pause가 false일때 (재생중)
            pause = true; // true로 바꾼후 play 멈춤
            stop();
        }
    }
}
```

```
class CDPlayer extends Player {
    // 부모 클래스의 추상 메소드를 구현
    void play(int currentPos) {
        // . . .
    }

    void stop(){
        // . . .
    }

    // CDPlayer 클래스에 추가로 정의된 멤버
    int currentTrack; // 현재 재생 중인 트랙

    void nextTrack() {
        currentTrack++;
        // . . .
    }

    void preTrack() {
        if (currentTrack > 1) {
            currentTrack--;
        }
        // . . .
    }
}
```


추상 클래스 (Abstract Class)

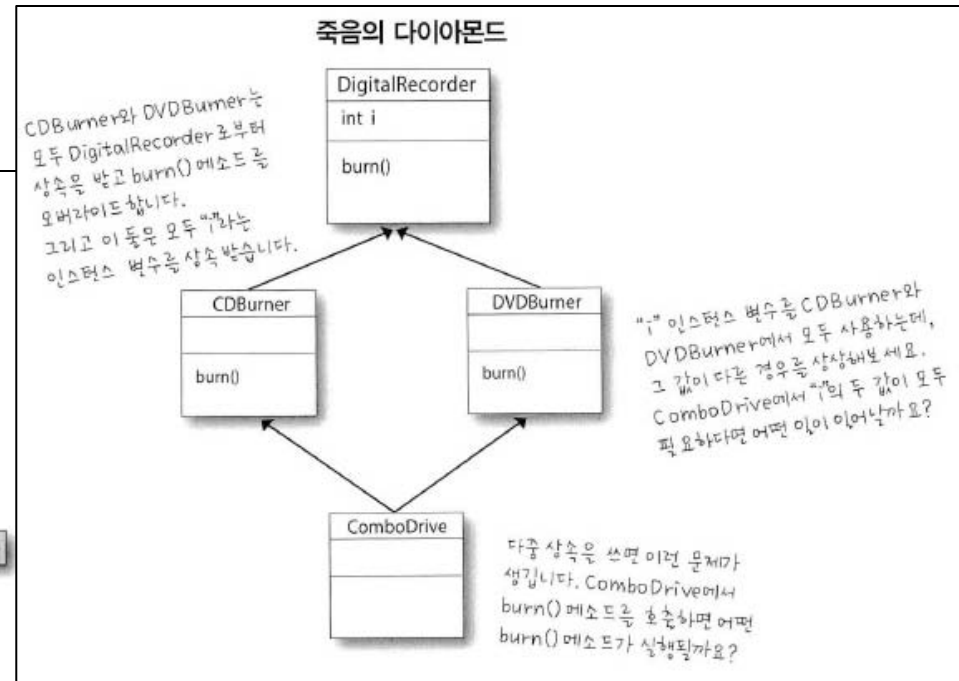
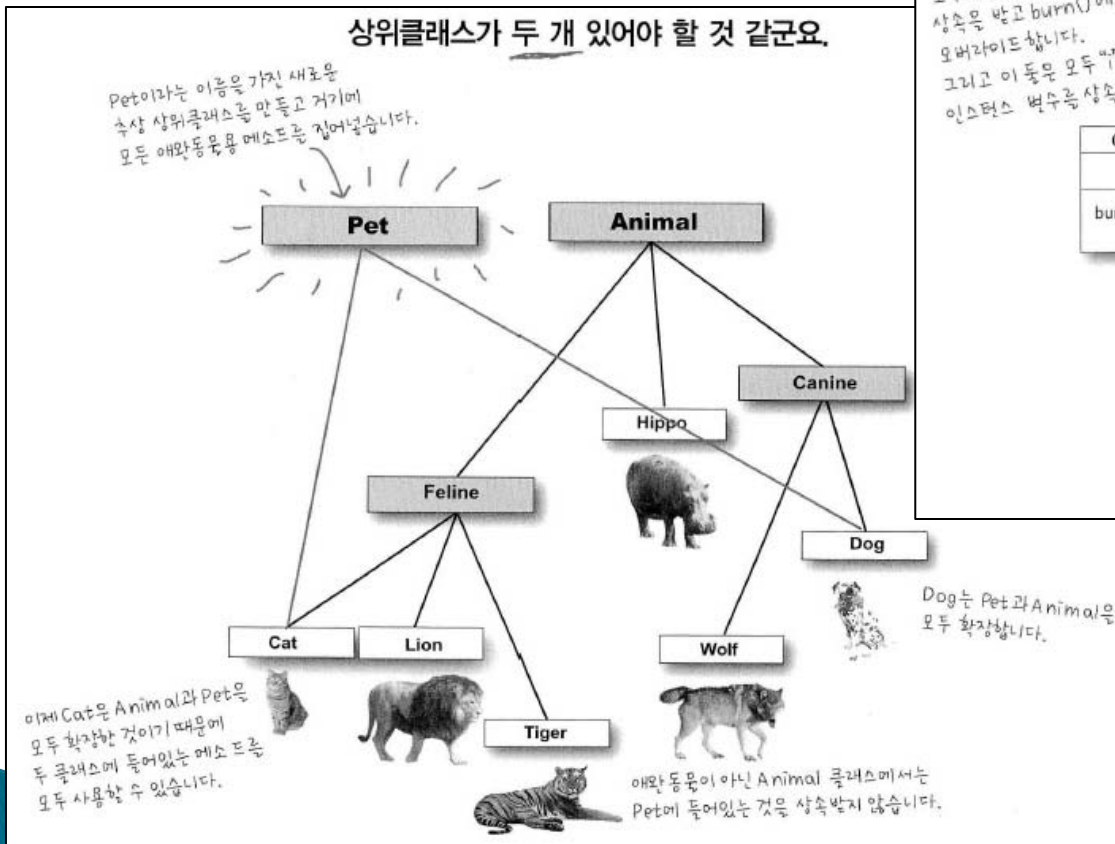
- ▶ 에이... 그냥 이렇게 써도 되잖아요?

```
class Player {  
    // . . .  
    void play(int pos) {}  
    void stop() {}  
    // . . .  
}
```

- ▶ 자손 클래스에서 반드시 구현하도록 강요
 - 이 땅 놀릴 거면 너 못 물려줘!
 - 저는 놀릴 수도 있는데 그럼 손자한테 맡길게요
 - 상속받은 자식 클래스가 추상 메소드를 구현하지 않으면 이 클래스 또한 추상 클래스가 된다.
- ▶ 순수한 추상 클래스
 - 미완성 설계도가 아니라 기본 설계도
 - 골격만 가지고 있는 클래스
 - 추상 메소드로만 이루어진 클래스
- ▶ 다중상속의 꿈 실현
 - 그러나 실제로 인터페이스를 통한 다중상속은 주된 이유가 아님
- ▶ 인스턴스 생성 불가

인터페이스 (Interface)

▶ Animal (re-revisit)



인터페이스 (Interface)

▶ 기본 사용법

```
interface InterfaceName {  
    public static final DataType VarName = Value;  
    public abstract MethodName (Input Parameter);  
}
```

- 모든 멤버변수는 `public static final` 이어야하며, 생략 가능
- 모든 메소드는 `public abstract` 이어야하며, 생략 가능

```
interface PlayingCard {  
    public static final int SPADE = 4;  
    final int DIAMOND = 3;    // public static final int DIAMOND = 3;  
    static int HEART = 2;    // public static final int HEART = 2;  
    int CLOVER = 1;          // public static final int CLOVER = 1;  
  
    public abstract String getCardNumber();  
    String getCardKind();    // public abstract String getCardKind();  
}
```

▶ 상속

```
interface Movable {  
    // 지정된 위치 (x,y)로 이동하는 기능의 메소드  
    void move (int x, int y);  
}  
  
interface Attackable {  
    // 지정된 대상(u)을 공격하는 기능의 메소드  
    void attack (Unit u);  
}  
  
interface Fightable extends Movable, Attackable { }
```

인터페이스 (Interface)

▶ 구현

```
class ClassName implements InterfaceName {  
    // 인터페이스에 정의된 추상 메소드 구현  
}  
  
class Fighter implements Fightable {  
    public void move(){ /* . . . */ }  
    public void attack(){ /* . . . */ }  
}
```

```
abstract class Fighter implements Fightable {  
    public void move() { /* . . . */ }  
}
```

```
class Fighter extends Unit implements Fightable {  
    public void move (int x, int y) { /* . . . */ }  
    public void attack (Unit u) { /* . . . */ }  
}
```

인터페이스 (Interface)

구현

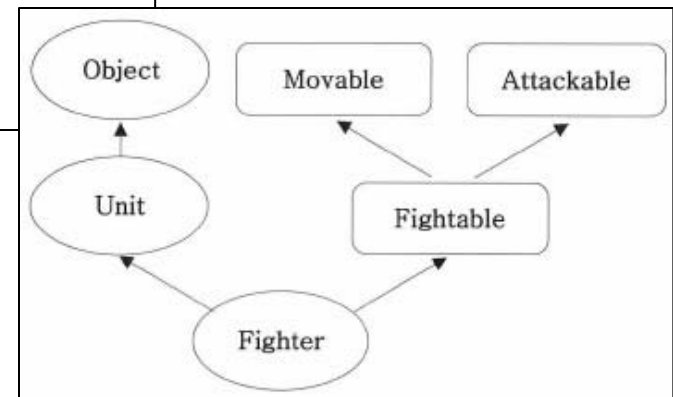
```
class FighterTest {  
    public static void main(String[] args){  
        Fighter f = new Fighter();  
  
        if (f instanceof Unit){  
            System.out.println("f는 Unit클래스의 자손입니다.");  
        }  
        if (f instanceof Fightable){  
            System.out.println("f는 Fightable 인터페이스를 구현했습니다.");  
        }  
        if (f instanceof Movable){  
            System.out.println("f는 Movable 인터페이스를 구현했습니다.");  
        }  
        if (f instanceof Attackable){  
            System.out.println("f는 Attackable 인터페이스를 구현했습니다.");  
        }  
        if (f instanceof Object){  
            System.out.println("f는 Object 클래스의 자손입니다.");  
        }  
    }  
}
```

f는 Unit클래스의 자손입니다.
f는 Fightable인터페이스를 구현했습니다.
f는 Movable인터페이스를 구현했습니다.
f는 Attackable인터페이스를 구현했습니다.
f는 Object클래스의 자손입니다.

```
interface Fightable extends Movable, Attackable { }  
interface Movable { void move(int x, int y); }  
interface Attackable { void attack(Unit u); }
```

```
class Fighter extends Unit implements Fightable {  
    public void move(int x, int y) { /* . . . */ }  
    public void attack(Unit u) { /* . . . */ }  
}
```

```
class Unit {  
    int currentHP; // 유닛의 체력  
    int x; // 유닛의 x좌표  
    int y; // 유닛의 y좌표  
}
```



인터페이스 (Interface)

▶ 다중상속

```
public interface IVCR {  
    public void play();  
    public void stop();  
    public void reset();  
    public int getCounter();  
    public void setCounter(int c);  
}
```

```
public class Tv {  
    protected boolean power;  
    protected int channel;  
    protected int volume;  
  
    public void power() { power = !power; }  
    public void channelUp() { channel++; }  
    public void channelDown() { channel--; }  
    public void volumeUp() { volume++; }  
    public void volumeDown() { volume--; }  
}
```

```
public class TVCR extends Tv implements IVCR {  
    VCR vcr = new VCR();  
  
    public void play(){  
        // 코드 작성 대신  
        // VCR 인스턴스의 메소드 호출  
        vcr.play();  
    }  
    public void stop(){  
        vcr.stop();  
    }  
    public void reset(){  
        vcr.reset();  
    }  
    public int getCounter(){  
        return vcr.getCounter();  
    }  
    public void setCounter(int c){  
        vcr.setCounter(c);  
    }  
}
```

```
public class VCR {  
    // VCR의 카운터  
    protected int counter;  
  
    public void play(){  
        // Tape 재생  
    }  
    public void stop(){  
        // 재생 멈춤  
    }  
    public void reset(){  
        counter = 0;  
    }  
    public int getCounter(){  
        return counter;  
    }  
    public void setCounter(int c){  
        counter = c;  
    }  
}
```

인터페이스 (Interface)

▶ 다중상속

```
/**
 * Queue 형태로 데이터를 삽입하고 추출하는 인터페이스
 */
public interface IQueue{
    void enqueue(String video);
    String dequeue();
}
```

```
/**
 * 저장소 역할을 하는 클래스
 */
import java.util.Vector;
public class Shop{
    protected Vector store = new Vector();
    public int getCount(){
        return store.size();
    }
}
```

```
/**
 * 보유한 비디오 개수는 0
 * 메트릭스1반납
 * 메트릭스2반납
 * 메트릭스3반납
 * 보유한 비디오 개수는 3
 * 메트릭스1 빌림
 * 메트릭스2 빌림
 * 메트릭스3 빌림
 * 보유한 비디오 개수는 0
 */
```

■ Shop과 IQueue를 동시에 상속 및 구현한 VideoShop 클래스

```
1 public class VideoShop extends Shop implements IQueue {
2     //...작업내용
3 }
```

```
/**
 * IQueue와 Shop을 동시에 상속하고 구현한 예
 */
public class VideoShop extends Shop implements IQueue{
    public void enqueue(String video){
        System.out.println(video + "반납");
        this.store.addElement(video);
    }
    public String dequeue(){
        return (String)this.store.remove(0);
    }
}
```

```
/**
 * VideoShop 클래스를 테스트하는 예
 */
public class VideoShopMain{
    public static void main(String[] args){
        String temp;
        VideoShop vs = new VideoShop();
        System.out.println("보유한 비디오 개수는 " + vs.getCount());

        //1. 3개의 비디오 반납
        vs.enqueue("메트릭스1"); //비디오 반납
        vs.enqueue("메트릭스2"); //비디오 반납
        vs.enqueue("메트릭스3"); //비디오 반납
        System.out.println("보유한 비디오 개수는 " + vs.getCount());

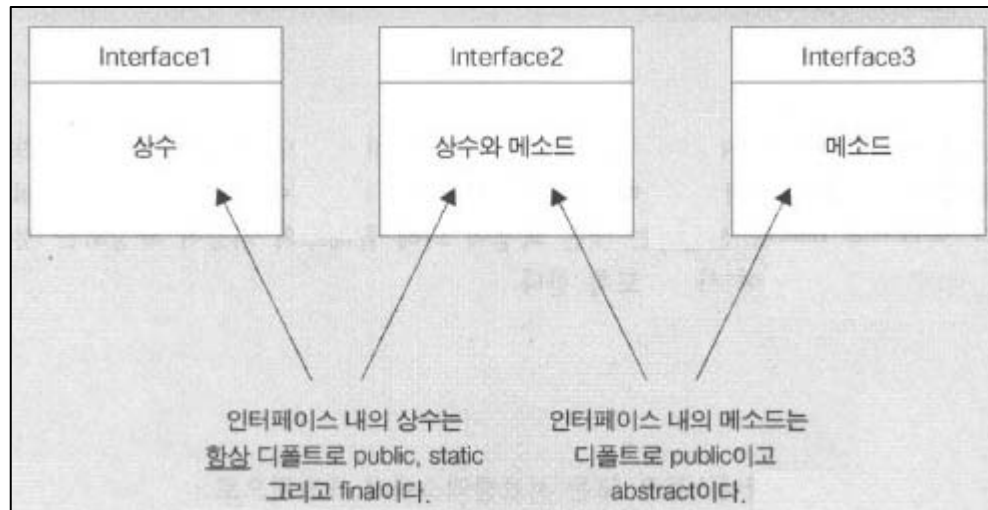
        //2. 3개의 비디오 빌려 줌
        temp = vs.dequeue(); System.out.println(temp + " 빌림");
        temp = vs.dequeue(); System.out.println(temp + " 빌림");
        temp = vs.dequeue(); System.out.println(temp + " 빌림");
        System.out.println("보유한 비디오 개수는 " + vs.getCount());
    }
}
```


인터페이스 (Interface)

▶ 장점

- 개발시간 단축
- 표준화 가능
- 서로 관계없는 클래스들에게 관계를 맺어 줄 수 있다.
- 독립적인 프로그래밍이 가능하다.

▶ 디폴트



인터페이스 (Interface)

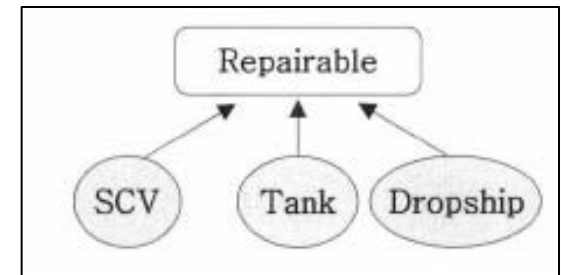
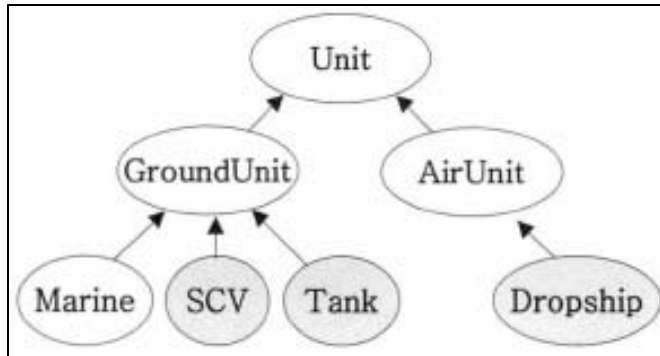
▶ 장점

```
void repair(Tank t){  
    // Tank를 수리  
}
```

```
void repair(Dropship d){  
    // Dropship 수리  
}
```

```
void repair(GroundUnit gu){  
    // 지상유닛 수리  
}
```

```
void repair(AirUnit au){  
    // 공중유닛 수리  
}
```



```
interface Repairable { }
```

```
class SCV extends GroundUnit implements Repairable { /* . . . */ }
```

```
class Tank extends GroundUnit implements Repairable { /* . . . */ }
```

```
class Dropship extends AirUnit implements Repairable { /* . . . */ }
```

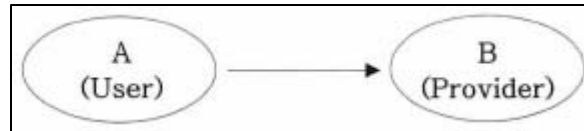
```
void repair(Repairable r){  
    // 유닛 수리  
}
```

인터페이스 (Interface)

▶ 결국 객체지향...

- 클래스를 사용하는 쪽 (User)와 클래스를 제공하는 쪽 (Provider)
- 메소드를 사용(호출) 하는쪽 (User)에서는
 - 사용하려는 메소드(Provider)의 선언부만 알면된다 (내용은 몰라도 된다)

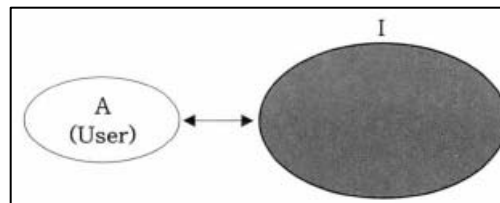
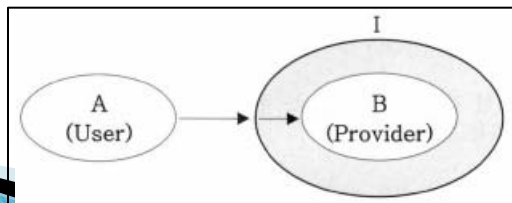
```
class A {  
    public void methodA(B b){  
        b.methodB();  
    }  
}  
  
class B {  
    public void methodB(){  
        System.out.println("methodB()");  
    }  
}  
  
class InterfaceTest{  
    public static void main(String args[]){  
        A a = new A();  
        a.methodA(new B());  
    }  
}
```



```
interface I {  
    public abstract void methodB();  
}
```

```
class B implements I {  
    public void methodB(){  
        System.out.println("methodB in B class");  
    }  
}
```

```
class A {  
    public void methodA(I i){  
        i.methodB();  
    }  
}
```



예외처리 (Exception Handling)

▶ Error VS Exception

□ **error** [ˈerə(r)]

1. [명사] C, U ~ in sth/in doing sth 실수[오류](특히 문제를 일으키거나 결과에 영향을 미치는 것)

No payments were made last week because of a computer error. ❏ 지난주에 컴퓨터 오류로 급여가 지급되지 못했다.

There are too many errors in your work. ❏ 당신이 해 놓은 일에는 실수가 너무 많아.

□ **exception** [ɪkˈsepʃn]

1. [명사](일반적인 상황의) 예외

Most of the buildings in the town are modern, but the church is an exception. ❏ 시내의 건물들은 대부분이 현대 건물들이지만 그 교회는 예외이다.

With very few exceptions, private schools get the best exam results. ❏ 극히 일부의 예외가 있긴 하지만 사립학교들이 가장 우수한 시험 성적을 낸다.

2. [명사](법칙을 따르지 않는) 사례 [예외]

Good writing is unfortunately the exception rather than the rule. ❏ 글을 잘 쓰는 것이 불행히도 일반적인 일이 아니라 이례적인 일이다.

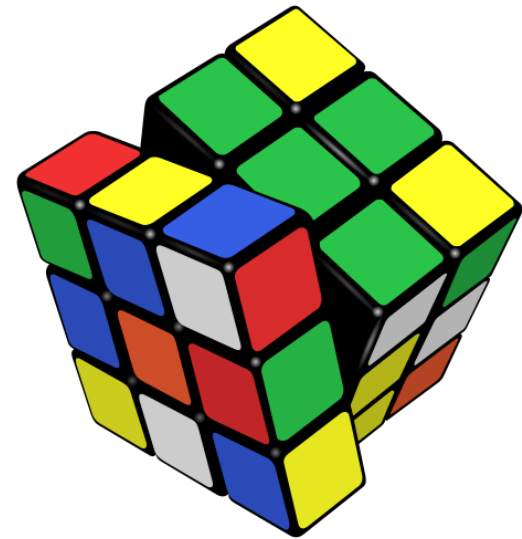
There are always a lot of exceptions to grammar rules. ❏ 문법 규칙들에는 언제나 예외적인 것들이 아주 많다.



예외처리 (Exception Handling)

▶ Java Error

- Error
 - 심각한
 - 프로그래머 범위 외
 - JVM
 - 비정상 종료
- Exception
 - 상대적으로 가벼운
 - 프로그래머 범위 내
 - 프로그래밍에 의존
 - 프로세스를 유지



예외처리 (Exception Handling)

- ▶ 자바의 에러
 - 컴파일 타임 에러 (Compile-Time Error)
 - 코드를 컴파일 할 때 발생하는 에러
 - 문법적 에러이므로 처리가 쉬움
 - 컴파일러가 기본적인 검사수행 (오타, 문법, 데이터타입)
 - 컴파일이 되지 않는 구문상의 오류
 - 런 타임 에러 (Run-Time Error)
 - 프로그램이 실행되고 있는 도중 발생하는 에러
 - 컴파일이 잘 되었어도 일어남
 - 프로그램 자체가 멈출 수 있음
 - 디버깅으로 잡을 수 밖에 없는 경우가 많음
 - 컴파일은 되지만 실행이 안되는 로직(Logic) 오류
- ▶ 자바 실행 시 (Runtime) 발생하는 오류
 - 에러(Error)
 - 발생하면 복구 불가능
 - 심각한 오류
 - 프로그램이 비정상 종료
 - 예외(Exception)
 - 발생하더라도 수습이 가능
 - 에러(error)에 비하면 덜 심각
 - 적절한 코드를 미리 작성해두면 비정상 종료 방지 가능

예외처리 (Exception Handling)

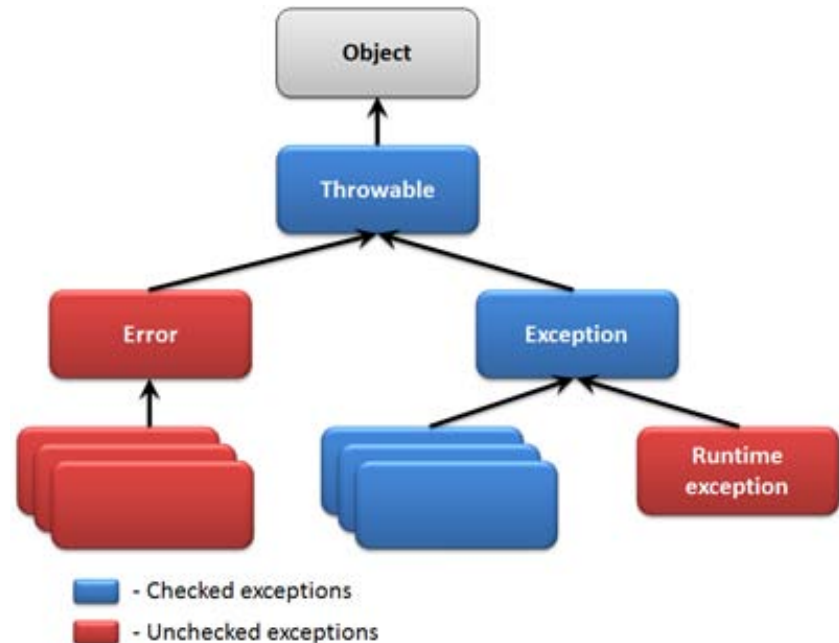
▶ Java Exception

◦ Compile-Time Exception

- 컴파일 시점
- 문법
- 컴파일러가 체크
- 컴파일 불가
- 상대적으로 쉬움

◦ Run-Time Exception

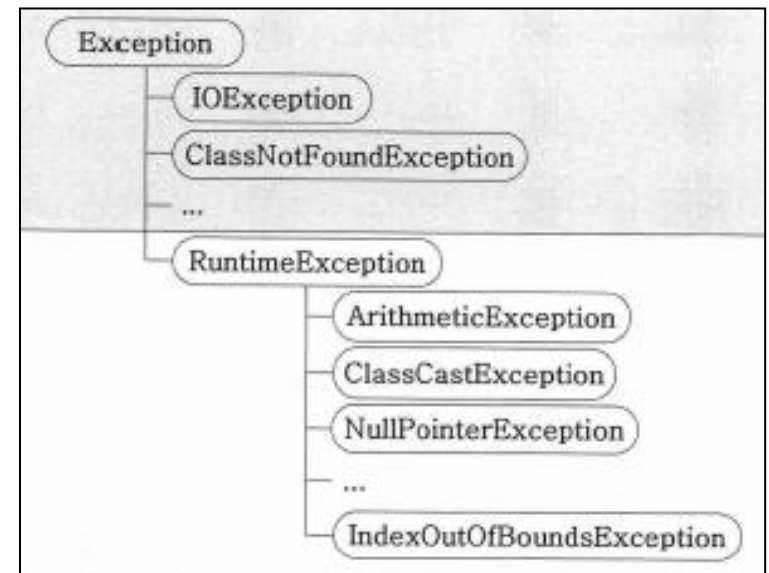
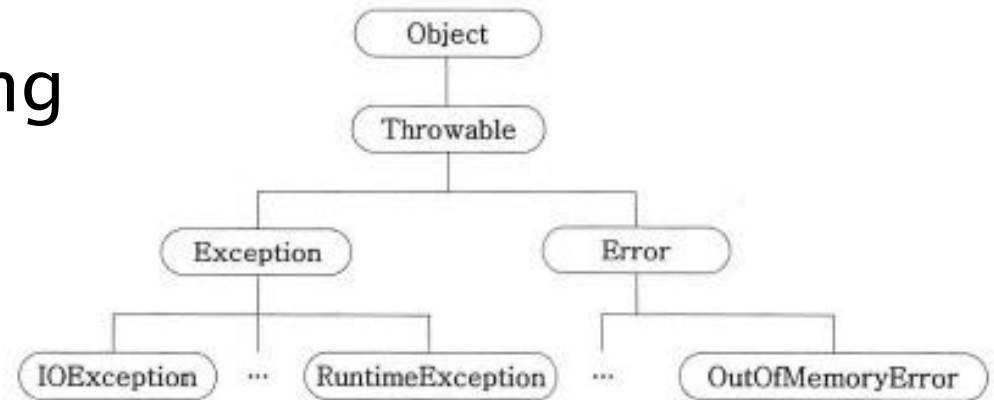
- 실행 시점
- 컴파일 완료
- 프로그램이 멈춤
- 디버깅
- 일반적으로 프로그램 로직 오류



예외처리 (Exception Handling)

▶ Exception Handling

- 종료 없이 계속 수행
- Exception is
 - Error Event
 - Object
 - Class for handling Exception



예외처리 (Exception Handling)

▶ Exception ?

- 실행 타임(Run-Time)에 발생하는 에러 이벤트(Error Event)
 - 에러 이벤트가 프로그래머가 처리할 부분
 - 프로그래머가 처리하지 않으면 가상머신이 이벤트를 받음
 - 가상머신의 경우는 대부분 프로그램을 중지시킴
- try-catch
 - Exception 감지, 처리 구문
 - C, C++ 과는 달리 Java에서는 에러 발생 확률이 높은 곳에도 강제로 사용하게 되어 있다.

▶ try-catch

```
try {  
    // 예외가 발생할 수 있는 Code  
} catch (Exception1 e1) {  
    // Exception1 발생시, 이를 처리하기 위한 Code  
} catch (Exception2 e2) {  
    // Exception2 발생시, 이를 처리하기 위한 Code  
...  
} catch (ExceptionN eN) {  
    // ExceptionN 발생시, 이를 처리하기 위한 Code  
}
```

try 블록내에서 예외가 발생하면?

- 1) 발생한 예외와 일치하는 catch 블록 검색
- 2) 일치하는 catch 블록이 있다면
 - 그 catch 블록 내 문장 수행후 전체 try-catch 문 빠져나감
 - 일치하는 catch 블록이 없을 경우 예외는 처리되지 못함

try 블록내에서 예외가 발생하지 않으면?

- 1) catch 블록을 거치지 않고 전체 try-catch문을 빠져나감

- 실행 타임 에러를 미연에 방지
- 에러 이벤트 발생시 발생 위치 확인
- 에러 이벤트 발생시 적절한 대처
- 프로그램의 비정상 종료를 막고, 정상적인 실행상태 유지

예외처리 (Exception Handling)

▶ Classification by Compile Handling

- Checked Exception

- 컴파일러가 명시적으로 예외처리를 요구
- 예외처리 안하면 컴파일 불가
- IOException, SQLException

- Unchecked Exception

- 컴파일러가 명시적으로 예외처리 요구하지 않음implicit
- 예외처리보다, 디버깅으로 해결
- RuntimeException



예외처리 (Exception Handling)

▶ Classification by Class

◦ Defined Exception

- Java API에서 제공된 Exception 클래스로 예외처리
- IOException, SQLException, RuntimeException...

◦ Custom Exception

- 개발자가 직접 만든 Exception 클래스로 예외처리
- 반드시 Exception 클래스를 상속받아야함



예외처리 (Exception Handling)

▶ Exception Handling Process

- 예외 발생 -> JVM에게 전달
 - JVM은 예외를 분석 -> 알맞은 Exception 클래스 객체 생성
 - 생성된 Exception 객체를 예외 발생한 코드로 전달
 - Exception 을 처리하거나, 처리하지 못할 경우 프로그램 종료
-
- try/catch
 - throws



예외처리 (Exception Handling)

▶ try ~ catch

```
try {  
    // 예외가 발생할 수 있는 Code  
} catch (Exception1 e1) {  
    // Exception1 발생시, 이를 처리하기 위한 Code  
} catch (Exception2 e2) {  
    // Exception2 발생시, 이를 처리하기 위한 Code  
...  
} catch (ExceptionN eN) {  
    // ExceptionN 발생시, 이를 처리하기 위한 Code  
}
```

try 블록 내에서 예외가 발생하면?

- 1) 발생한 예외와 일치하는 catch 블록 검색
- 2) 일치하는 catch 블록이 있다면
 - 그 catch 블록 내 문장 수행후 전체 try-catch 문 빠져나감
 - 일치하는 catch 블록이 없을 경우 예외는 처리되지 못함

try 블록 내에서 예외가 발생하지 않으면?

- 1) catch 블록을 거치지 않고 전체 try-catch문을 빠져나감



예외처리 (Exception Handling)

```
class ExcpetionEx1 {  
    public static void main(String args[]){  
        int number = 100;  
        int result = 0;  
  
        for ( int i=0 ; i<10 ; i++ ){  
            result = number / (int) (Math.random() * 10);  
            System.out.println(result);  
        }  
    }  
}
```



```
class ExceptionEx2 {  
    public static void main(String args[]){  
        int number = 100;  
        int result = 0;  
  
        for ( int i=0 ; i<10 ; i++ ){  
            try {  
                result = number / (int) (Math.random() * 10);  
                System.out.println(result);  
            } catch (ArithmeticException e) {  
                System.out.println("0");  
            } // try-catch 끝  
        } // for 끝  
    }  
}
```


예외처리 (Exception Handling)

▶ Multiple Catch

- 여러 개 Exception
- 순차적
- 적합한 Exception 찾기
- 자식에서 부모로



```
try{  
    //실행문1;  
    //실행문2;  
    //실행문3;  
    //실행문4;  
  
}catch( 예외처리 클래스1 변수1){  
    //예외처리1  
}  
}catch( 예외처리 클래스2 변수2){  
    //예외처리2  
}  
}catch( 예외처리 클래스3 변수3){  
    //예외처리3  
}  
}catch( 예외처리 클래스4 변수4){  
    //예외처리4  
}  
}
```

```
public class ExceptionTest{  
  
    public static void main( String [ ]args){  
  
        try{  
            int [ ] num = new int[2];  
            num[0] = 1;  
            num[1] = 2;  
            num[2] = 3;  
            System.out.println("배열 123" );  
        }catch( ArrayIndexOutOfBoundsException e){  
            System.out.println("ArrayIndexOutOfBoundsException 처리" );  
        }catch( Exception e){  
            System.out.println("Exception 처리" );  
        }  
        System.out.println("정상종료" );  
    }  
}
```

예외처리 (Exception Handling)

▶ 다단계 catch

```
/**
 * catch 블록으로 처리하지 못한 경우
 * NullPointerException 블록이 없기 때문에 에러 발생
 */
```

```
import java.io.*;
public class LevelErrorMain{
    public static void main(String[] args){
        try{
            FileReader f = new FileReader("LevelErrorMain.java");
            String s = null;
            System.out.println(s.toString()); //NullPointerException 발생
        }catch(FileNotFoundException e1){
            System.out.println("FileNotFoundException:" + e1);
        }catch(ArrayIndexOutOfBoundsException e2){
            System.out.println("ArrayIndexOutOfBoundsException:" + e2);
        }
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
    at LevelErrorMain.main(LevelErrorMain.java:10)
```

```
/**
 * Exception으로 모든 에러 처리
 */
```

```
import java.io.*;
public class LevelCatchMain{
    public static void main(String[] args){
        try{
            FileReader f = new FileReader("LevelCatchMain.java");
            String s = null;
            System.out.println(s.toString()); //NullPointerException 발생
        }catch(FileNotFoundException e1){
            System.out.println("FileNotFoundException:" + e1);
        }catch(ArrayIndexOutOfBoundsException e2){
            System.out.println("ArrayIndexOutOfBoundsException:" + e2);
        }catch(Exception e3){
            System.out.println("Exception:" + e3);
        }
    }
}
```

예외처리 (Exception Handling)

▶ 다단계 catch

catch의 단계별 사용

```
1 try{
2     //작업
3 } catch(FileNotFoundException e1){
4     //FileNotFoundException 예러처리
5 } catch(ArrayIndexOutOfBoundsException e2){
6     //ArrayIndexOutOfBoundsException 예러처리
7 } catch(Exception e3){
8     //그 외의 모든 예러
9 }
```

```
FileNotFoundException e1 = x; //불가능
ArrayIndexOutOfBoundsException e2 = x; //불가능
Exception e3 = x; //가능
```

모든 예러 이벤트에 대한 처리

```
1 try{
2     //...작업
3 } catch(Exception e){
4     //...모든 예러 이벤트 처리
5 }
```

catch(Exception e3)과 유사한 else문

```
1 if(조건){...}
2 else if(조건){...}
3 else if(조건){...}
4 else if(조건){...}
5 else{....} //그 외의 경우
```

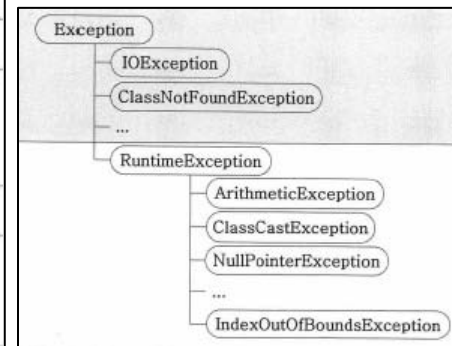
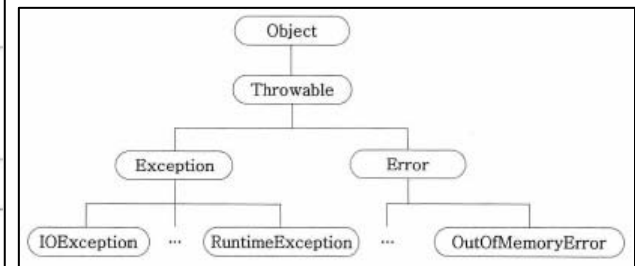
catch의 순서가 잘못된 경우(잘못된 catch의 단계별 사용)

```
1 try{
2     //....작업
3 } catch(Exception e3){ //모든 예러를 처리하기 때문에 하위의 catch 구문은 의미가 없어진다.
4     //....예러처리 코드
5 } catch(FileNotFoundException e1){
6     //....예러처리 코드
7 } catch(ArrayIndexOutOfBoundsException e2){
8     //....예러처리 코드
9 }
```

예외처리 (Exception Handling)

▶ 예외 클래스 계층 구조

클래스명	나타내는 예외 상황
ArithmeticException	정수를 0으로 나누려고 하는 등의 유효하지 않은 계산 조건을 사용하는 경우
IndexOutOfBoundsException	객체의 범위를 벗어난 인덱스를 사용하려고 하는 경우. 배열, String 객체, 또는 Vector 객체가 이에 해당한다. Vector 클래스는 표준 패키지 java.util에 정의되어 있다.
NegativeArraySizeException	음의 크기를 갖는 배열을 정의하려 하는 경우
NullPointerException	null을 포함하는 객체 변수를 사용하려는 경우. 적당한 작업을 위해서는 변수가 객체를 참조해야 한다. 예를 들어, 메소드를 호출하거나 데이터 멤버에 접근하는 경우가 이에 속한다.
ArrayStoreException	배열 타입에 맞지 않는 객체를 배열에 저장하려는 경우
ClassCastException	객체를 부적절한 타입으로 형변환하려 한 경우. 즉, 객체가 지정한 클래스도 아니고, 지정한 클래스의 슈퍼클래스나 서브클래스도 아닌 경우를 뜻한다.
IllegalArgumentException	메소드가 파라미터 타입과 일치하지 않는 인자를 전달하는 경우
SecurityException	프로그램이 보안에 위반되는 부적절한 작업을 수행하려는 경우, 애플릿에서 로컬 컴퓨터에 있는 파일을 읽으려 하는 경우가 이에 속한다.
IllegalMonitorStateException	스레드가 자기가 소유하지 않은 객체를 모니터링하려 할 때
IllegalStateException	적절하지 않은 때에 메소드를 호출하는 경우
UnsupportedOperationException	객체가 지원하지 않는 작업을 수행하도록 요청하는 경우



- RuntimeException클래스와 그 자손클래스들
- Exception클래스와 그 자손클래스들

RuntimeException클래스들 - 프로그래머의 실수로 발생하는 예외
 Exception클래스들 - 사용자의 실수와 같은 외적인 요인에 의해 발생하는 예외

예외처리 (Exception Handling)

▶ Example

```
class ExcpetionEx1 {
    public static void main(String args[]){
        int number = 100;
        int result = 0;

        for ( int i=0 ; i<10 ; i++ ){
            result = number / (int) (Math.random() * 10);
            System.out.println(result);
        }
    }
}
```

```
20
100
java.lang.ArithmeticException: / by zero
    at ExceptionEx2.main(ExceptionEx2.java:7)
```

```
16
20
11
0 ← ArithmeticException이 발생해서 0이 출력되었다.
25
100
25
33
14
12
```

```
class ExceptionEx2 {
    public static void main(String args[]){
        int number = 100;
        int result = 0;

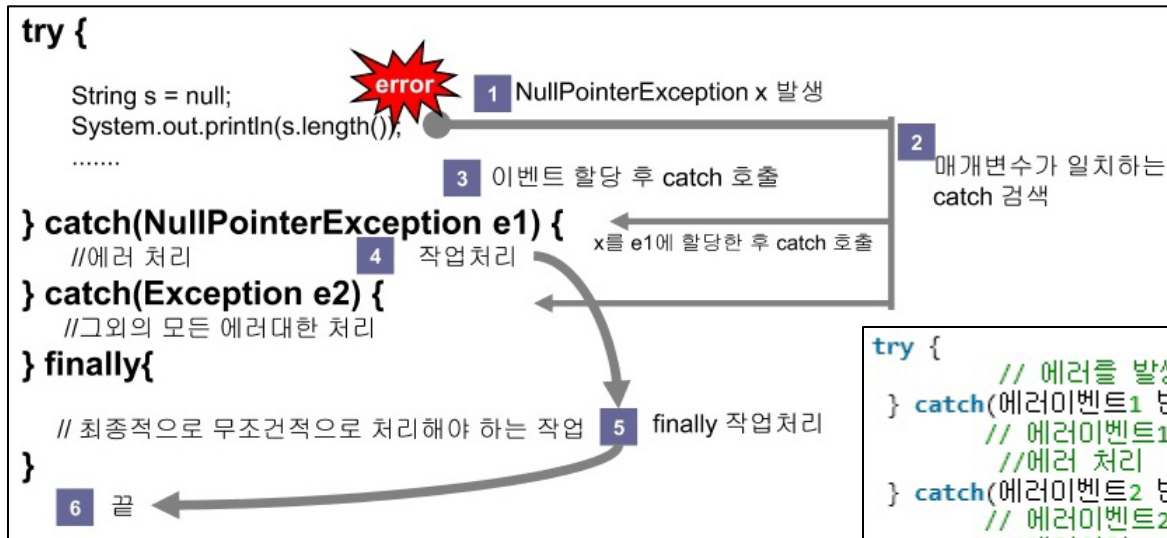
        for ( int i=0 ; i<10 ; i++ ){
            try {
                result = number / (int) (Math.random() * 10);
                System.out.println(result);
            } catch (ArithmeticException e) {
                System.out.println("0");
            } // try-catch 끝
        } // for 끝
    }
}
```

```
import java.io.*;
public class ErrorMain{
    public static void main(String[] args) {
        FileReader fr = new FileReader("ErrorMain.java");
    }
}
```

```
import java.io.*;
public class ErrorMain2{
    public static void main(String[] args){
        try{
            FileReader fr = new FileReader("ErrorMain.java");
            //fr을 이용해서 작업
        }catch(FileNotFoundException e){
            System.out.println(e);
        }
    }
}
```


예외처리 (Exception Handling)

▶ try-catch-finally



```
try {  
    // 에러를 발생할 가능성이 있는 코드  
} catch(에러이벤트1 변수){  
    // 에러이벤트1이 발생했을 때 이벤트가 catch로 넘어오게 된다.  
    //에러 처리  
} catch(에러이벤트2 변수){  
    // 에러이벤트2가 발생했을 때 이벤트가 catch로 넘어오게 된다.  
    //에러처리  
} finally{  
    // 최종적으로 무조건적으로 처리해야 하는 작업  
}
```

■ 기본적인 에러처리 순서

1. **try** 블록을 일단 한번 시도한다.
2. **try** 블록 내에서 에러가 발견되면 에러 이벤트가 발생한다.
3. **catch** 블록에 일치하는 이벤트가 있다면 해당 **catch** 블록으로 간다.
4. 해당 **catch** 블록을 호출해서 에러처리를 한다. (여기서 다시 **try** 블록으로 되돌아갈 수 없다.)
5. 에러가 발생하든 하지 않은 **finally**는 무조건 실행한다.

예외처리 (Exception Handling)

```
public class ExceptionTest{  
    public static void main( String []args){  
        try{  
            int [] num = new int[2];  
            num[0] = 1;  
            num[1] = 2;  
            System.out.println("1 2 입력");  
        } catch( ArrayIndexOutOfBoundsException e){  
            System.out.println("ArrayIndexOutOfBoundsException 에러발생");  
        } finally {  
            System.out.println("에러발생 유무와 무관하게 반드시 수행된다");  
        }  
        System.out.println("정상 종료");  
    }  
}
```

```
public class ExceptionTest{  
    public static void main( String []args){  
        try{  
            int [] num = new int[2];  
            num[0] = 1;  
            num[1] = 2;  
            num[2] = 3;  
            System.out.println("1 2 3 입력");  
        } catch( ArrayIndexOutOfBoundsException e){  
            System.out.println("ArrayIndexOutOfBoundsException 에러발생");  
        } finally {  
            System.out.println("에러발생 유무와 무관하게 반드시 수행된다");  
        }  
        System.out.println("정상 종료");  
    }  
}
```


예외처리 (Exception Handling)

```
/**
 * 실행 시에 NullPointerException 예외가 발생하는 예
 */
public class NullErrorMain {
    public static void main(String[] args) {
        String str = null;
        System.out.println(str.length()); //에러발생
        System.out.println("프로그램의 종료");
    }
}
```

```
/**
 * 예외처리 루틴이 삽입된 예(실행 시에 예외처리 루틴에 의해 예외가 처리된다.)
 */
public class TryCatchMain{
    public static void main(String[] args) {
        try{
            String str = null;
            System.out.println(str.length());
        }catch(NullPointerException e){
            System.out.println(e.toString() + " 예외가 발생했습니다");
            System.out.println("예외처리 루틴 실행");
        }
        System.out.println("프로그램의 종료");
    }
}
```

```
/**
 * 예외처리 루틴(try, catch, finally)이 삽입된 예
 * finally 블록의 무조건적인 실행
 */
public class TryCatchFinallyMain{
    public static void main(String[] args) {
        try{
            String str = null;
            System.out.println(str.length());
        }catch(NullPointerException e){
            System.out.println(e.toString() + " 예외가 발생했습니다");
            System.out.println("예외처리 루틴 실행");
        }finally{
            //예외가 나든 나지 않은 무조건 실행되는 블록
            System.out.println("finally 구문 실행");
        }
        System.out.println("프로그램의 종료");
    }
}
```

예외처리 (Exception Handling)

▶ 수동으로 예외 발생시키기

1. `new` 연산자를 이용하여 발생시키려는 예외 클래스의 객체를 만들
`Exception e = new Exception("고의로 발생시켰음");`
2. `throw` 키워드를 이용하여 예외 발생시킴
`throw e;`

```
class ExceptionEx9 {  
    public static void main(String args[]){  
        try {  
            Exception e = new Exception("고의로 발생시켰음.");  
            throw e; // 예외를 발생시킴  
            // 한줄로 쓸수도 있다.  
            // throw new Exception("고의로 발생시켰음.");  
        } catch (Exception e) {  
            System.out.println("에러 메시지 : " + e.getMessage());  
            e.printStackTrace();  
        }  
        System.out.println("프로그램이 정상 종료되었음.");  
    }  
}
```

에러 메시지 : 고의로 발생시켰음.
java.lang.Exception: 고의로 발생시켰음.
 at ExceptionEx9.main(ExceptionEx9.java:6)
프로그램이 정상 종료되었음.

```
--Exception 발생구문--  
정보:e.getMessage(): 이것이 에러 메시지  
정보:e.toString(): java.lang.Exception: 이것이 에러 메시지  
java.lang.Exception: 이것이 에러 메시지  
    at UseThrowMain.main(UseThrowMain.java:7)  
finally: 결국이리로 오는군요
```

```
public class UseThrowMain {  
    public static void main(String args[]) {  
        try {  
            throw new Exception("이것이 에러 메시지");  
        } catch (Exception e) {  
            System.out.println("--Exception 발생구문--");  
            System.out.println("정보:e.getMessage(): " + e.getMessage());  
            System.out.println("정보:e.toString(): " + e.toString());  
            e.printStackTrace();  
            return;  
        } finally{  
            System.out.println("finally: 결국이리로 오는군요");  
        }  
    }  
}
```

예외처리 (Exception Handling)

▶ 메소드 예외 선언

```
void method() throws Exception1, Exception2, ... , ExceptionN {  
    // . . .  
}
```

java.lang
Class InterruptedException

java.lang.Object
+-- java.lang.Throwable
+-- java.lang.Exception
+-- java.lang.InterruptedException

All Implemented Interfaces:
[Serializable](#)

wait

```
public final void wait()  
    throws InterruptedException
```

Causes current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object. In other words this method behaves exactly as if it simply performs the call `wait(0)`.

The current thread must own this object's monitor. The thread releases ownership of this monitor and waits until another thread notifies threads waiting on this object's monitor to wake up either through a call to the `notify` method or the `notifyAll` method. The thread then waits until it can re-obtain ownership of the monitor and resumes execution.

This method should only be called by a thread that is the owner of this object's monitor. See the `notify` method for a description of the ways in which a thread can become the owner of a monitor.

Throws:
[IllegalMonitorStateException](#) - if the current thread is not the owner of the object's monitor.
[InterruptedException](#) - if another thread has interrupted the current thread. The *Interrupted status* of the current thread is cleared when this exception is thrown.

See Also:
[notify\(\)](#), [notifyAll\(\)](#)

```
class ExceptionEx18 {  
    public static void main(String[] args) throws Exception {  
        method1(); // 같은 클래스내 static 멤버이므로 객체 생성없이 호출  
    }  
    static void method1() throws Exception {  
        method2();  
    }  
    static void method2() throws Exception {  
        throw new Exception();  
    }  
}
```

```
java.lang.Exception  
    at ExceptionEx18.method2(ExceptionEx18.java:12)  
    at ExceptionEx18.method1(ExceptionEx18.java:8)  
    at ExceptionEx18.main(ExceptionEx18.java:4)
```

- 예외가 발생했을 때, 모두 3개의 메서드(main, method1, method2)가 호출스택에 있었으며,
- 예외가 발생한 곳은 제일 윗줄에 있는 method2()라는 것과
- main메서드가 method1()를, 그리고 method1()은 method2()를 호출했다는 것을 알 수 있다.

예외처리 (Exception Handling)

▶ 예외 메시지 보기

```
public static int divide(int[] array, int index) {
    try {
        System.out.println("\nmain()의 첫 번째 try 블록 실행");
        array[index + 2] = array[index] / array[index + 1];
        System.out.println("divide()의 첫 번째 try 블록 실행");
        return array[index + 2];
    } catch (ArithmeticException e) {
        System.err.println("divide()에서 ArithmeticException이 발생했다 \n" +
            "\n예외 객체의 메시지:\n\t" +
            e.getMessage());
        System.err.println("\n스택 기록 출력:\n");
        e.printStackTrace();
        System.err.println("\n스택 기록 출력 종료\n");
    } catch (ArrayIndexOutOfBoundsException e) {
        System.err.println("divide()에서 ArrayIndexOutOfBoundsException이 발생했다\n" +
            "\n예외 객체의 메시지:\n\t" + e.getMessage());
        System.err.println("\n스택 기록 출력:\n");
        e.printStackTrace();
        System.out.println("\n스택 기록 출력 종료\n");
    } finally {
        System.err.println("divide()의 finally절");
    }
    System.out.println("divide()의 try 블록의 다음 코드 실행");
    return array[index + 2];
}
```

메소드	설명
getMessage()	이 메소드는 현재 예외를 설명하는 메시지 내용을 리턴한다. 이는 대부분 예외 클래스(Throwable의 서브클래스)의 완전한 이름과 간단한 예외 설명으로 이루어진다.
printStackTrace()	이 메소드는 메시지와 스택 기록을 표준 에러 출력 스트림(콘솔 프로그램의 경우에는 화면)으로 출력한다.
printStackTrace(PrintStream s)	이 메소드는 출력 스트림을 인자로 지정한다는 점을 제외하고는 printStackTrace() 메소드와 동일하다. 예외 객체 e의 이전 메소드 호출은 다음과 같다. e.printStackTrace(System.err);

Exception

▶ throws

- method call mechanism
- 여기서 Exception 처리 안해...
 - 자신을 호출한 곳으로 발생한 예외를 떠넘김
 - 적어도 main()에서는 try/catch 처리를 해야함
- RuntimeException 계열을 throw 할 필요없음
- 메소드 선언시 throws Exception 클래스를 지정
 - public void method() throws Exception{ . . . }

```
main() {  
    try{  
        a();  
    }catch(발생Exception e){  
        e.printStackTrace();  
    }  
}  
  
void a() throws 발생Exception{  
    b();  
}  
  
void b() throws 발생Exception{  
    // 예외발생  
}
```

Exception

```
public class Throw{  
  
    public void go() { //throws Exception {  
  
        System.out.println("go() 시작");  
  
        int [ ] num = new int[2];  
        num[0] = 1;  
        num[1] = 2;  
        num[2] = 3;  
        System.out.println("1 2 3 입력");  
    }  
}
```

```
public class ThrowTest{  
    public static void main(String[] args){  
        Throw t= new Throw();  
  
        try{  
  
            t.go(); //이리로 Exception 이 넘어 온다.  
        }catch(Exception e){  
  
            System.out.println("main() : catch(Exception) " + e);  
        }  
        System.out.println("정상종료!!");  
    }  
}
```


Exception

- ▶ 명시적 Exception Class
 - 강제로 Exception 발생
 - 사용자 정의 Exception 사용
 - 필요에 의한 Exception 사용
 - throw new ExceptionClass();
 - 반드시 try/catch로 예외처리 해주어야함

```
import java.io.IOException;
public class Throw{

    public void go() throws IOException {

        System.out.println("go() 시작");

        int [] num = new int[2];
        num[0] = 1;
        num[1] = 2;

        if( num.length == 2 ) throw new IOException("배열크기가 2" );
    }
}
```

```
import java.io.IOException;
public class ThrowTest{
    public static void main(String[] args){
        Throw t= new Throw();

        try{

            t.go(); //이리로 Exception 이 넘어 온다.

        }catch(IOException e){

            System.out.println("main() : catch(IOException) " + e);
        }

        System.out.println("정상종료!!");
    }
}
```


Exception

▶ Custom Exception

- 프로그래머가 제작
- Exception 상속
- Exception 명시적으로 발생시킴

```
public class UserDefineException extends Exception{  
  
    public UserDefineException( String msg){  
        super( msg );  
    }  
}
```

```
public class ThrowTest{  
    public static void main(String[] args){  
        Throw t= new Throw();  
  
        try{  
  
            t.go(); //이리로 Exception 이 넘어 온다.  
        }catch(UserDefineException e){  
  
            System.out.println("main() : catch(UserDefineException) " + e);  
        }  
        System.out.println("정상종료!!");  
    }  
}
```

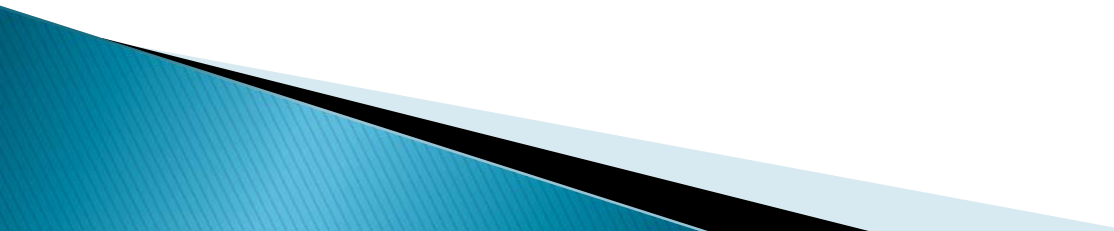
```
public class Throw{  
  
    public void go() throws UserDefineException {  
  
        System.out.println("go() 시작");  
  
        int [] num = new int[2];  
        num[0] = 1;  
        num[1] = 2;  
  
        if( num.length == 2) throw new UserDefineException("배열크기가 2");  
    }  
}
```

그 외 다루지 않은 부분

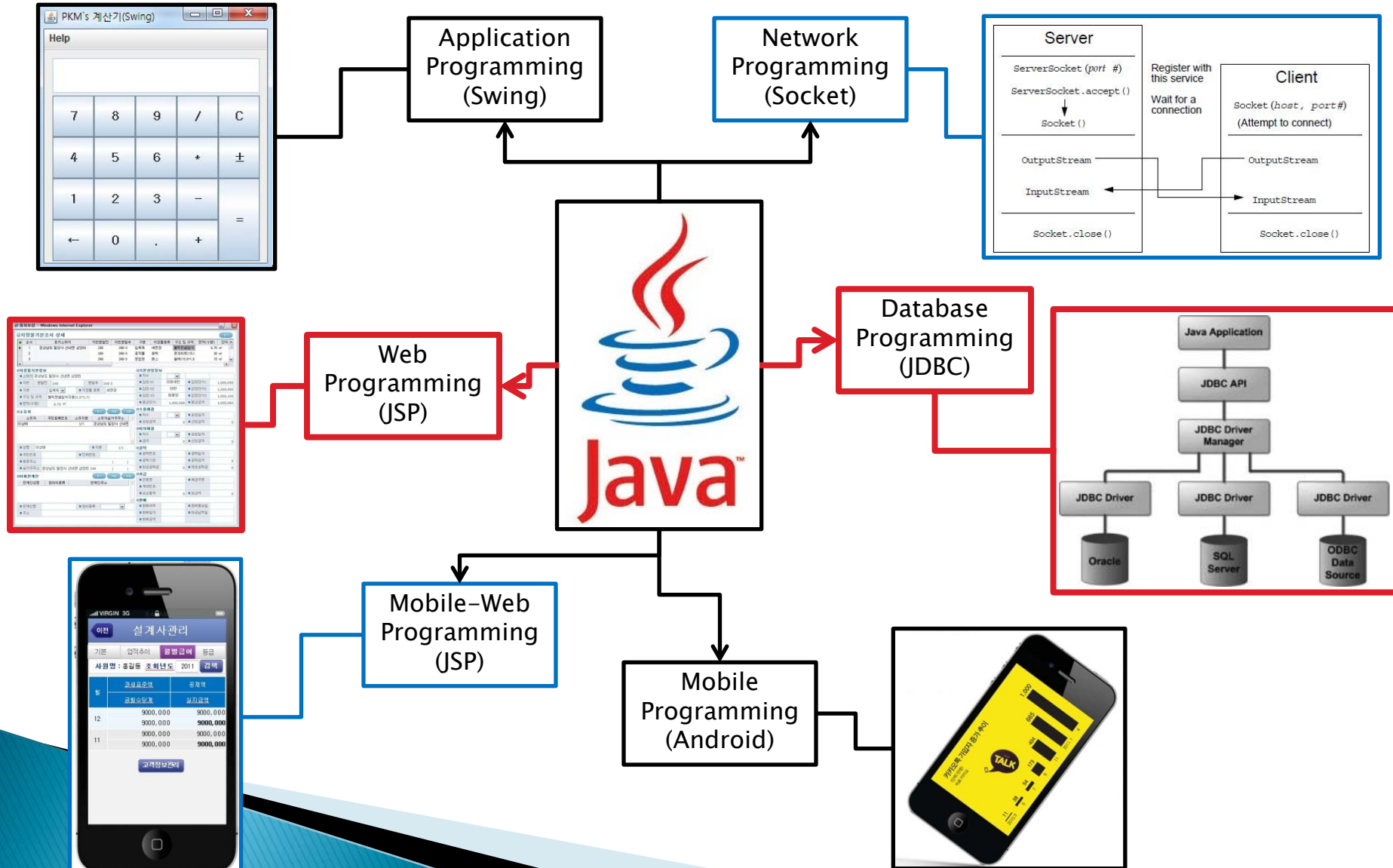
▶ 키워드

- Object Class
 - equals, hashCode, toString, clone, finalize, wait, notify, getClass
- Math Class
- Wrapper Class
- Inner Class
- Calendar, Date Class
- Random Class

그 외 다루지 않은 부분

- ▶ 키워드
 - 쓰레드 (Thread)
 - Collection API
 - Swing
 - I/O (Stream), File
 - Networking (Socket Programming)
 - Generics
 - Serialization
- 

그 외 다루지 않은 부분



오늘 숙제

- ▶ 다형성을 비롯한 객체지향의 개념은
 - 수업 한번, 혹은 책 한번 훑어보는 것으로는 이해하기가 쉽지 않다.
 - 개념적으로나 구현적으로 쉽지 않으므로,
 - 반드시 복습 + 코딩을 해볼 것
- ▶ 숙제로 만들었던 **Animal** 클래스를
 - 수업 때 설명한 방식으로 추상화/구상화
- ▶ 클래스, 추상클래스, 인터페이스를 만든 후
 - 각각을 extends, implements 하되
 - 모든 클래스, 인터페이스에 동일한 메소드를 만들어놓고
 - 최후 구현클래스에서 동일한 메소드 명으로 만들 때,
 - 구현이 적용되는 우선순위를 알아볼 것