



클 래 스



# 클래스의 구조

- 클래스 선언부에 올 수 있는 것
  - 주석문
  - 패키지
  - 임포트
  - 클래스 선언
- 클래스 내용부에 올 수 있는 것
  - 멤버변수(member variable), field, 속성(attribute), property
  - 메소드
  - 내부클래스



# 클래스

- ❖ 모든 객체들의 생산처
- ❖ 클래스 = 객체를 생성하는 틀
- ❖ 프로그래밍이 쓰이는 목적을 생각하여 어떤 객체를 만들어야 하는지 결정한다.
- ❖ 각 객체들이 어떤 특징(속성과 동작)을 가지고 있을지 결정한다.
- ❖ 객체들 사이에서 메시지를 주고 받도록 만들어 준다.

# 객체의 구성

- ❖ 속성(Attribute) - 멤버변수
- ❖ 동작(Behavior) - 메소드

```
class TV {  
    int channel;  
    int volumn;  
    public void channelUp() {  
    }  
    public void channelDown() {  
    }  
}
```

# 추상화와 클래스

❖ 필요한 객체를 설계해서 프로그램이 인식하게 하는 방법

- 클래스를 설계한다.
- 클래스로부터 객체를 생성한다.
- 생성된 객체는 클래스에 정의한 속성

```
class TV {  
    int channel;  
    int volumn;  
    public void channelUp() {  
    }  
    public void channelDown() {  
    }  
}  
TV tv = new TV();  
tv.channelDown();
```

# 클래스의 선언

public / default

final / abstract

[접근제한자] [활용제한자] **class** 클래스명 {

속성 정의 (멤버변수)

기능 정의 (메소드)

}

❖ 객체(object / instance) 와 인스턴스 변수 ??



메 소 드

# 메소드

- 메소드(Method)

객체가 할 수 있는 행동을 정의

메소드의 이름은 소문자로 시작하는 것이 관례

public / protected / default() / private



static / final / abstract / synchronized



[접근제한자] [활용제한자] 리턴타입 메소드이름([매개변수들]) {

행위 기술.....

[return 값;]

}

public static void main(String [] a) { }





# 메소드

- 프로그램에 필요한 데이터는 변수
- 프로그램에 대한 필요한 로직은 메서드
  - 로직의 이름은 동사로 시작
  - 로직의 이름을 결정한 후에는 로직의 파라미터 결정
  - 로직의 파라미터 결정 후에는 로직의 리턴타입 결정



# 메소드

- 메소드 선언

- 선언시 { } 안에 메소드가 해야 할 일을 정의
- 메서드의 선언은 반드시 클래스 블록 사이에 정의 되어야 한다.

```
class Test {  
    void call(int val) {  
        실행 코드 나열  
        .....  
    }  
}
```

# 메소드

- 메소드 호출

- 호출 단계

- 1) 호출하려는 메소드가 선언되어 있는 클래스 객체를 생성한다.

- `Test t = new Test( );`

- 2) 클래스객체.메소드 이름으로 호출한다.

- `t.call( 100 );`

```
class Test {  
    void call( ) {  
        .....  
    }  
}
```

# 메소드

- **static** 이 메소드에 선언되어 있을 때는 객체 생성 없이 클래스 이름으로 직접 접근하는 방법을 권장함.

클래스이름.메소드 이름으로 호출

=====

```
Test t = new Test( );
```

```
t.call( );
```

-----

```
Test.call( );
```

→ 권장

```
class Test {  
    static void call( ) {  
        .....  
    }  
}
```

# 메소드

- 매개변수
  - 메소드 선언 시 사용하는 것
- 인자
  - 호출하는 쪽에서 전달하는 것

```
public void soundUp ( int val ) {  
    // 이 위치에서 선언되는 것  
    String msg = “지역변수”;  
}
```

선언

```
=====  
soundUp(10);
```

호출

# 메소드

- 메서드의 4가지 유형

1. 반환 값이 없고 매개변수가 없음

```
void call ( ) {
```

```
.....
```

```
}
```

선언

=====

```
call ( );
```

호출

# 메소드

2. 반환 값이 없고 매개변수가 있음

```
void call ( int val ) {
```

```
.....
```

```
}
```

선언

=====

```
call ( 10 );
```

```
int i = 10;
```

```
call ( i );
```

호출

# 메소드

## 3. 반환 값이 있고 매개변수가 없음

```
double call ( ) {  
    return 1.1;  
}
```

선언

=====

```
double d = call ( );
```

호출



# 메소드

## 4. 반환 값이 있고 매개변수가 있음

```
int square ( int val ) {  
    return val * val;  
}
```

선언

=====

```
int result = call ( 10 );
```

호출



# 메서드 주요 사항

# 메소드

- 메소드에서 받은 매개변수는 그 메소드에서 선언한 지역 변수와 똑같이 간주 됩니다.
- 메소드에 매개변수가 있으면 반드시 해당 유형의 값을 전달해야만 합니다.

```
public void soundUp( int val ) {  
    // 이 위치에서 선언되는 것  
    String msg = “지역변수”;  
}  
=====
```

soundUp(10);

지역변수는??

# 메소드

- 메소드로부터 값을 받을 수도 있습니다.
- 리턴 유형은 메소드를 선언할 때 지정하며, 리턴 유형이 정해져 있으면 반드시 그 유형의 값을 리턴해야만 합니다.

```
void volumnUp(int val) {  
    // 이 위치에서 선언되는 것  
    String msg = “지역변수”;  
}
```

```
=====  
volumnUp(10);
```

```
int getVolumn( ) {  
    return 10;  
}
```

```
=====  
int volumn = getVolumn( ); ( O )
```

# 메소드

- 메소드에 여러 개의 인자를 전달할 수 있습니다.
- 메소드에는 값이 전달되는 방식을 사용합니다. ( 기본형, 참조형 )

- 선언

```
public void init(int channel, int volume) {  
}
```

- 호출

```
int channel = 7;    int volume = 15;  
init( channel, volume );
```

channel

7

volume

15

# 메소드

- 만약, 여러 개의 값을 리턴하고 싶다면?? 배열이용, Collection 객체 이용

- 선언

```
public String[] multiArr( ){  
    String [] arr = { "a", "b" };  
    return arr;  
}
```

=====

- 호출

```
String [] result = multiArr( );
```

# 메소드


- 만약, 리턴값의 유형이 double로 선언했다면, double이 받을 수 있는 값들은 모두 리턴이 가능합니다.

```
public double getVolumn( ) {  
    return 10;  
}
```

- 리턴값을 무조건 사용해야 하는 것은 아닙니다.

```
public double getVolumn( ) {  
    return 10;  
}
```

```
int result = getVolumn(); (X)  
double result = getVolumn(); (O)  
getVolumn(); (O)
```



# 처음 메서드 사용시 주로 발생하는 오류





# 메소드

## 1. 메서드 선언시 반환타입을 생략하면 오류임 : 반환값이 없을 경우 **void** 선언

- 오류발생

```
call ( ) {  
    System.out.println ("오류발생");  
}
```

- 정상

```
void call ( ) {  
    System.out.println ("오류발생");  
}
```

## 2. 반환타입이 **void**가 아니라면 반드시 메서드 내에 **return** 문이 있어야 함

- 오류발생

```
int plus ( int i, int k ) {  
    System.out.println ( “두수의 합 : ” + ( i + k ) );  
}
```

- 정상

```
int plus ( int i, int k ) {  
    System.out.println ( “두수의 합 : ” + ( i + k ) );  
    return i + k;  
}
```

### 3. 반환타입이 **void** 일 경우 선택적으로 **return** 문을 사용할 수 있음 **return** 문을 사용할 경우 값을 설정하면 안됨

- 정상

```
void process ( ) {
```

```
}
```

```
void process ( ) {
```

```
    return ;
```

```
}
```

## 4. 메서드의 매개변수 선언 시 `int i, int k` 라고 하지 않고 `int i, k` 형식으로 사용하는 경우

- 오류발생

```
void plus ( int i, k ) {  
    System.out.println ("두수의 합 : " + ( i + k ) );  
}
```

- 정상

```
void plus ( int i, int k ) {  
    System.out.println ("두수의 합 : " + ( i + k ) );  
}
```

## 5. 메서드의 매개변수 선언 시 값을 설정하는 경우

- 오류발생

```
void call ( 100 ) {  
    System.out.println ( 100 );  
}
```

- 정상

```
void call ( int i ) {  
    System.out.println ( “넘어온 값 : ” + i );  
}
```

## 6. 메서드의 선언 시에 매개변수 오른쪽 괄호를 닫은 후에 세미콜론을 사용 시 오류 발생

- 오류 발생

```
void plus ( int i, k );
```

- 정상

```
void plus ( int i, int k ) {  
    System.out.println ("두수의 합 : " + ( i + k ) );  
}
```

## 7. 메서드의 매개변수에 선언된 변수를 메서드 내에 다시 선언하는 경우

- 오류발생

```
void process ( int i) {  
    int i = 200;  
    System.out.println ( i );  
}
```

- 정상

```
void process ( int i) {  
    System.out.println ( i );  
}
```

## 8. 매개변수에 선언된 변수의 타입과 맞지 않는 인자를 넘겨주는 경우

### - 오류발생

```
선언 : void print ( String s ) {  
        System.out.println ( s );  
    }
```

```
호출 : print ( 100 );
```

### - 정상

```
선언 : void print ( String s ) {  
        System.out.println ( s );  
    }
```

```
호출 : print ( “^^” );
```





# Java 문자열과 API 사용법



# 1. 문자열 정의

- ➔ 자바에서는 문자열을 객체로 취급
- ➔ `java.lang` 패키지에 포함
- ➔ `java.lang.String`, `StringBuffer`, `StringBuilder` 클래스 제공
- ➔ `String` 클래스 : 한번 생성된 다음 변하지 않는 문자열에 사용
- ➔ `StringBuffer` 클래스 : 계속하여 변할 수 있는 문자열에 사용, 동기화 적용
- ➔ `StringBuilder` 클래스 : 계속하여 변할 수 있는 문자열에 사용, 비동기화



## construct

String()


String(char chars[])

String(char chars[], int startindex, int numChars)

String(String strObj)

String(byte asciiChars[])

String(byte asciiChars[], int startIndex, int numChars)



## 문자열 길이


`int length()` : 문자열의 길이를 반환

## 문자열 추출

`char charAt(int i)` : 문자열 중에서 *i*번째 문자를 반환

`void getChars(int sourceStart, int sourceEnd, char target[],  
int targetStart)`

➔ 문자열의 일부를 문자 배열로 (`target[]`) 제공



## 문자 비교

`boolean equals(Object str)` : `str`로 지정된 문자열과 현재의 문자열 같은지 비교

`boolean equalsIgnoreCase(String str)` : 문자열 비교시 대소문자 무시

`boolean startsWith(String str)` : 문자열이 `str`로 시작하면 `true`, 아니면 `false`

`boolean endsWith(String str)` : 문자열이 `str`로 끝나면 `true`, 아니면 `false`

`int compareTo(String str)` : 현재의 문자열과 `str`로 지정된 문자열을 비교하여 현재의 문자열이 `str`로 지정된 문자열보다 크면 양수, 같으면 0, 작으면 음수값을 반환.

작다는 의미는 순서(알파벳)에 따라 앞에 온다는 의미



## 문자열 탐색

`int indexOf(String str)`

`int indexOf(String str, int startIndex)`

`int lastIndexOf(String str)`

`int lastIndexOf(String str, int startIndex)`

## 문자열 변환

`String substring(int startIndex, int endIndex)`      부분 문자열을 반환

`String concat(String constr)`      결합된 문자열 반환


`String replace(char original, char replacement)`      치환된 문자열 반환

`String trim()` 문자열의 시작과 끝부분에 있는 공백이 제거된 문자열을 반환

`String toLowerCase()` 소문자로 반환

`String toUpperCase()` 대문자로 반환

`String [] split(String regexp)` regexp를 기준으로 문자열을 나눈다.



## 형의 변환

`static String valueOf(double num)`

`static String valueOf(long num)`

`static String valueOf(Object obj)`

`static String valueOf(char chars[])`

`static String valueOf(char chars[], int startIndex, int numChars)`



생 성 자





# 생성자 특징

1. 클래스 명과 이름이 동일
2. 반환타입이 없다.

```
public class Dog {  
    Dog ( ) {  
        System.out.println("나는 생성자 입니다.");  
        System.out.println("클래스와 이름이 동일하고 반환타입이 없어요");  
    }  
}
```

# 생성자 특징

## 3. 디폴트 생성자

- 클래스내에 생성자가 하나도 정의되어 있지 않을 경우  
**JVM**이 자동으로 제공하는 생성자
- 형태 : 매개변수가 없는 형태, 클래스명() {}

```
class Dog {
```

```
    생성자가 하나도 없는 상태임
```

```
    JVM 이 자동으로 제공함
```

```
    ??????
```

```
    Dog( ) { }
```

```
}
```

```
class Main {
```

```
    public static void main(String [] a) {
```

```
        // 객체 생성
```

```
        Dog d = new Dog( );
```

```
    }
```

```
}
```

# 생성자 특징

## 4. 오버로딩을 지원한다.

- 클래스 내에 메소드 이름이 같고 매개변수의 타입 또는 개수가 다른 것

```
class Dog {  
    Dog( ) { }  
    Dog(String name) { }  
    Dog(int age) { }  
    Dog(String name, int age) { }  
}
```

```
class Main {  
    public static void main(String [] a) {  
        Dog d = new Dog( );  
        Dog d2 = new Dog("짱");  
        Dog d3 = new Dog(3);  
        Dog d4 = new Dog("메리", 4);  
    }  
}
```

# 생성자 특징

## 5. 객체를 생성할 때 속성의 초기화를 담당하게 한다.

```
class Dog {
```

```
    String name;
```

```
    int age;
```

```
    Dog( String n, int a) {
```

```
        name = n;
```

```
        age = a;
```

```
    }
```

```
}
```

```
class Main {
```

```
    public static void main(String [] a) {
```

```
        Dog d = new Dog( );
```

```
        d.name = “짱”;
```

```
        d.age = 3;
```

```
        Dog d2 = new Dog(“메리”, 4);
```

```
    }
```

```
}
```

# 생성자 특징

6. this의 활용 : static 영역에서는 사용이 불가능하다.

- **this.멤버변수**

- **this ( [ 인자값.. ] )** : 생성자 호출

- **this** 생성자 호출시 제한사항

  - : 생성자 내에서만 호출이 가능함

  - : 생성자 내에서 첫번째 구문에 위치해야 함

```
class Dog {  
    String name;  
    int age;  
    void info ( ) {  
        System.out.print(this.name);  
        System.out.println(this.age);  
    }  
}
```

# 생성자 특징

6. this의 활용 : static 영역에서는 사용이 불가능하다.

- **this**.멤버변수

- **this ( [ 인자값.. ] )** : 생성자 호출

- **this** 생성자 호출시 제한사항

  - : 생성자 내에서만 호출이 가능함

  - : 생성자 내에서 첫번째 구문에 위치해야 함

```
class Dog {  
    String name;  
    int age;  
    Dog ( ) {  
        this("짱");  
    }  
    Dog ( String name ) {  
    }  
}
```

# 문제. 1

다음의 프로그램이 실행 될 수 있도록 클래스를 작성하세요

```
public class ExDateMain {  
    public static void main(String[] args) {  
        ExDate ex01 = new ExDate();  
        ex01.showDate();  
  
        ExDate ex02 = new ExDate(2013, 8, 11);  
        ex02.showDate();  
    }  
}
```

출력결과

-----  
2013년 4월 01일  
2013년 8월 11일

## 문제. 2

다음의 프로그램이 실행 될 수 있도록 클래스를 작성하세요

```
public class DogMain {  
    public static void main(String[] args) {  
        Dog d = new Dog();  
        Dog d2 = new Dog("짱");  
        Dog d3 = new Dog(3);  
        Dog d4 = new Dog("메리", 2);  
  
        d.info();  
        d2.info();  
        d3.info();  
        d4.info();  
    }  
}
```

### 출력결과

```
-----  
강아지의 이름은 이름모름이고 나이는 0입니다.  
강아지의 이름은 짱이고 나이는 0입니다.  
강아지의 이름은 이름모름이고 나이는 3입니다.  
강아지의 이름은 메리이고 나이는 2입니다.
```





static



# static 특징

## 1. 로딩시점

- **static** : 클래스 로딩 시
- **nonStatic** : 객체 생성시

## 2. 메모리상의 차이

- **static** : 클래스당 하나의 메모리 공간만 할당
- **nonStatic** : 인스턴트 당 메모리가 별도로 할당

# static 특징

## [실행 시 메모리 영역]

타입 정보

1. Type Information
2. Constant Pool
3. Field Information
4. Method Information
5. Class Variables

Instance  
(객체 생성시)

PC Registers

Java Virtual  
Machine  
Stacks

Method  
Area  
(공유)

Heap  
(독립적)

Native  
Method  
Stacks

# static 특징

## 3. 문법적 특징

- **static** : 클래스 이름으로 접근
- **nonStatic** : 객체 생성 후 접근

```
class Employee {  
    static int empCount;  
    String name;  
}
```

```
class Main {  
    public static void main(String [] a) {  
        Employee e = new Employee( );  
        e.name = "손오공";  
        Employee.empCount ++;  
    }  
}
```

# static 특징

## 4. static 영역에서는 non-static 영역을 집적 접근이 불가능

```
class Main {  
    String name = "길동이";  
    public static void main(String [] a) {  
        System.out.println( name );  
    }  
}
```

오류발생



# static 특징

- non-static 영역에서는 static 영역에 대한 접근이 가능

```
class Main {  
    static int count = 100;  
    public void call () {  
        System.out.println( count );  
    }  
}
```



상 속

1. 확장성, 재 사용성
2. 클래스 선언 시 **extends** 키워드를 명시

```
class Employee {  
  
}  
class Manager extends Employee {  
  
}
```

3. 관계
  - 부모 (상위, **Super**) 클래스 : **Employee**
  - 자식 (하위, **Sub**) 클래스 : **Manager**



4. 자식 클래스는 부모 클래스에 선언 되어 있는 멤버변수, 메소드를 자신의 것처럼 사용할 수 있다.

단, 접근 제한자에 따라 사용 여부가 달라진다.

```
class Employee {  
    int    no;  
    String name;  
}
```

```
class Manager extends Employee {  
    public void info( ) {  
        System.out.println( no );  
        System.out.println( name );  
    }  
}
```

## 5. **super** 키워드

- **super.변수** : 상위 클래스의 멤버변수 접근
- **super.메소드명** : 상위 클래스의 메소드 접근
- **super ([인자, ..])** : 상위 클래스 생성자 호출

## 5. super 키워드

```
public class Employee {  
    String no;  
    String name;  
    String grade;  
    int salary;  
    public Employee(String no, String name,  
        String grade, int salary) {  
        this.no = no;  
        this.name = name;  
        this.grade = grade;  
        this.salary = salary;  
    }  
}
```

## 5. super 키워드

```
public class Manager extends Employee {  
    String no;  
    String name;  
    String grade;  
    int salary;  
    Employee [] subEmpList;  
    public Manager(String no, String name, String grade,  
                    int salary, Employee [] subEmpList) {  
        super(no, name, grade, salary);  
        this.subEmpList = subEmpList;  
    }  
}
```

## 6. 오버라이딩 ( 재정의 )

- 상속관계 에서 발생
- 상위클래스의 메소드를 하위클래스에서 동일한 메소드 선언을 하는 것.  
내용부를 다르게 정의함.
- 메소드 선언 부분에서 접근 제한자 부분은 다를 수 있다.  
단, 하위클래스 접근 제한자가 상위클래스 접근제한자 보다 좁으면 안된다.  
접근범위 >

**public > protected > default > private**

## 6. 오버라이딩 ( 재정의 )

```
public class Employee {  
    String no;  
    String name;  
    String grade;  
    int salary;  
    public void info() {  
        System.out.println("번호 : " + no);  
        System.out.println("이름 : " + name);  
        System.out.println("직책 : " + grade);  
        System.out.println("급여 : " + salary);  
    }  
}
```

## 6. 오버라이딩 ( 재정의 )

```
public class Manager extends Employee {  
    public void info() {  
        System.out.println("번호 : " + no);  
        System.out.println("이름 : " + name);  
        System.out.println("직책 : " + grade);  
        System.out.println("급여 : " + salary);  
        System.out.println("-----");  
        System.out.println("관리하는 사원들의 정보");  
        System.out.println("-----");  
        for (int i = 0; i < subEmpList.length; i++) {  
            subEmpList[i].info();  
        }  
        System.out.println("-----");  
    }  
}
```

## 6. 오버라이딩 ( 재정의 )

```
public class Manager extends Employee {  
    public void info() {  
        super.info();  
  
        System.out.println("-----");  
        System.out.println("관리하는 직원들의 정보");  
        System.out.println("-----");  
        for (int i = 0; i < subEmpList.length; i++) {  
            subEmpList[i].info();  
        }  
        System.out.println("-----");  
    }  
}
```



## 7. 오버로딩과 오버라이딩의 차이점

구분	오버로딩	오버라이딩
메소드 이름	동일	동일
매개변수	다름	동일
리턴타입	상관없음	동일

## 8. 상속과 생성자

- 모든 객체는 생성될 때 부모의 정보를 메모리에 생성
- 컴파일시 생성자와 관련된 처리
  1. 생성자에 **this()** 또는 **super()** 호출 코드가 있는지 확인
  2. 호출코드가 없다면 생성자의 첫 번째 명령어로 **super( )** 를 호출하는 코드가 자동으로 삽입됨

## 9. Object

- 모든 객체들이 자동으로 상속받는 클래스
- **extends** 를 명시적으로 사용하지 않은 경우 자동으로 **Object**를 상속받는 코드가 추가됨

ex> class Test → class Test extends java.lang.Object

- 결론 : 모든 클래스는 **Object** 에 있는 메서드를 사용할 수 있음



# 접근제한자



# 접근제한자

---

- **public** : 모든 위치에서 접근이 가능
- **protected** : 같은 패키지에서 접근이 가능, 다른 패키지 접근 불가능  
단, 다른 패키지의 클래스와 상속관계가 있을 경우 접근 가능
- **default** : 같은 패키지에서만 접근이 허용  
접근제한자가 선언이 안 되었을 경우 기본 적용
- **private** : 자신 클래스에서만 접근이 허용
  
- 클래스(외부) 사용가능 : **public, default**
- 내부클래스, 멤버변수, 메소드 사용 가능 : 4 가지 모두 가능

# 접근제한자

```
public class Test {  
    public    String name;  
    protected String pass;  
             String addr;  
    private  int money;  
}
```

kr.co.bit.pack.a

```
class A extends Test {  
    ???  
}
```

```
import kr.co.bit.pack.a.Test;  
class Main {  
    Test t = new Test ( );  
    t.??  
}
```


kr.co.bit.pack.b

```
import kr.co.bit.pack.a.Test;  
class B extends Test {  
    ???  
}
```

# 클래스도

```
class Person {  
    public String name;  
    protected int age;  
    String addr;  
    private String pass;  
  
    public void info ( ) { }  
    public String getAddr ( ) { .... }  
    public void setAddr ( String addr ) {  
        ....  
    }  
}
```

Person
+ name : String # age : int addr : String - pass : String
+ info( ) + getAddr( ) : String + setAddr( addr : String )



# 객체의 형변환



# 형변환

1. 정의 : "=" 연산자를 기준으로 좌변과 우변의 데이터 타입이 다른 경우에 발생
2. 조건 : 좌변과 우변의 객체가 상속 관계가 있어야 함

```
public class Printer { }  
public class LGPrinter { }  
Printer p = new LGPrinter (); (오류)  
=====
```

```
public class Printer { }  
public class LGPrinter extends Printer { }  
Printer p = new LGPrinter ();
```



# 묵시적 형변환

## 3. 종류

### 1) 묵시적 형변환

- 상위(부모) 클래스 타입 = 하위(자식) 클래스 타입

**Printer p = new LGPrinter ( );**

- 부모타입은 자식타입을 포함
- 기억해야 할 중요 포인트:

형변환된 상위 클래스 변수가 사용 할 수 있는 범위는 자신 클래스에 정의된 변수와 메소드만 사용이 가능

단, 상위클래스의 메소드를 하위클래스에서 오버라이딩 (재정의) 했을 경우 하위클래스에 선언된 메소드가 호출

# 묵시적 형변환

```
public class TV {  
    boolean power;  
    public void powerOn() {}  
}  
public class LgTV extends TV {  
    String name = "LGTV";  
    public void powerOn() {  
        power = true;  
    }  
    public void print() {  
        System.out.println(name);  
    }  
}
```

```
class Main {  
    public static void main(String [] a) {  
        TV tv = new LgTV( );  
        tv.power ;  
        tv.name;  
        tv.print( );  
        tv.powerOn( );  
    }  
}
```

O  
X  
X  
O



# 명시적 형변환

## 2) 명시적 형변환

- 형변환 연산자를 이용해서 변환
- 하위클래스 타입 = (하위클래스타입) 상위 클래스 타입

**LGPrinter lg = (LGPrinter) p;**

- 기억해야 할 중요 포인트 :

상위클래스 타입 자리에 올 수 있는 객체는 실제 가리키는 메모리가 하위 클래스 타입 이어야 가능

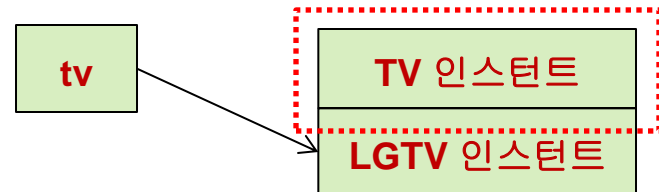
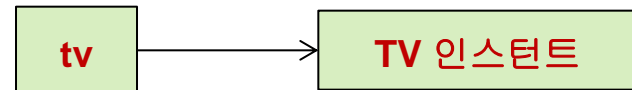
# 명시적 형변환

```
public class TV { }  
public class LgTV extends TV { }
```

```
LgTV lg = new TV(); // 컴파일 오류
```

```
TV tv = new TV( );  
LgTV lg = (LgTV) tv; // 실행시 오류
```

```
TV tv = new LgTV( );  
LgTV lg = (LgTV) tv; // 성공
```





# 추상 클래스 (abstract)

# 추상클래스

**abstract** : 클래스, 메소드, 멤버변수(X)

## 1. 추상클래스

- **abstract** 를 클래스 선언 부분에 추가함

```
abstract class Animal {  
}
```

## 2. 추상메소드

- **abstract** 를 메소드 선언 부분에 추가함
- 메소드의 선언부만 있고 바디( { } ) 가 없음

```
void print ( ) {  
}
```



```
abstract void print ( ) ;
```

# 추상클래스

3. 추상 메소드를 포함하는 클래스는 반드시 추상클래스로 선언되어야 함

```
class Test {  
    abstract void print();  
}
```



```
abstract class Test {  
    abstract void print( );  
}
```

4. 추상클래스는 인스턴스 생성이 불가능함. (new 키워드 사용 불가능)

```
abstract class Test { }  
class Main {  
    Test t = new Test( );  
}
```



오류발생



# 추상클래스

5. 추상클래스는 일반(구현된) 메소드와 추상 메소드 모두 선언이 가능함.


```
abstract class Printer {  
    abstract void print();  
    public void call () {  
        System.out.println("구현된 메소드");  
    }  
}
```

# 추상클래스

6. 추상클래스를 상속받는 하위클래스는 상위클래스의 추상 메소드를 반드시 오버라이딩(재정의) 해야 한다.

```
abstract class Printer {  
    abstract void print();  
    public void call () {  
        System.out.println("구현");  
    }  
}
```

```
class LGPrinter extends Printer {  
    void print( ) {  
        System.out.println("재정의");  
    }  
}
```



# 추상클래스

## 7. 추상클래스의 객체변수는 하위클래스를 이용함.

```
abstract class Printer { }  
class LGPrinter extends Printer { }  
-----  
class Main {  
    public static void main(String [ ] args) {  
        Printer p = new LGPrinter ( );  
    }  
}
```



# 인 터 페 이 스 (interface)

# 정의 및 특징

- 완벽한 추상화된 객체
- 반쯤 완성된 객체
- 설계도

## 1. **interface** 키워드를 이용하여 선언

```
interface ServerConstants { }
```

## 2. 선언되는 변수는 모두 상수로 적용

```
interface ServerConstants {  
    String SERVER_IP = "127.0.0.1";  
}
```

→ **public static final** String SERVER\_IP

# 특징

## 3. 선언되는 메소드는 모두 추상 메소드로 적용

```
interface Test {  
    void print( ) ;
```

```
    public void call ( ) {
```

```
    }
```

```
}
```



```
public abstract void print( ) ;
```



오류발생

# 특징

## 4. 객체 생성이 불가능 (추상클래스 동일한 특성)

```
interface Test { }
```

```
class Main {
```

```
    Test t = new Test( );
```

```
}
```

오류발생

## 5. 클래스가 인터페이스를 상속 할 경우에는 **extends** 키워드가 아니라 **implements** 키워드를 이용

```
interface shape {
```

```
}
```

```
class Circle implements Shape {
```

```
}
```

오류발생

# 특징

6. 인터페이스를 상속받는 하위클래스는 추상 메소드를 반드시 오버라이딩 (재정의) 해야 한다.

```
interface Printer {  
    void print( );  
}
```

```
class LGPrinter implements Printer {
```

```
    public void print( ) {  
        System.out.println("재정의");  
    }  
}
```

오류발생



## 7. 인터페이스 객체변수는 하위클래스를 이용함.

```
interface Printer { }  
class LGPrinter implements Printer { }  
-----  
class Main {  
    public static void main(String [ ] args) {  
        Printer p = new LGPrinter ( );  
    }  
}
```



final

# final

## final 3가지 사용법

1. 변수 : 상수
2. 메소드 : 오버라이딩 금지
3. 클래스 : 상속 금지

```
class Printer {  
    public final void print () {}  
}
```

```
class LGPrinter extends Printer {  
    public void print () {  
        System.out.println ("재정의");  
    }  
}
```

오류발생

# final

## final 3가지 사용법

1. 변수 : 상수
2. 메소드 : 오버라이딩 금지
3. 클래스 : 상속 금지

오류발생

```
final class String {  
    public void print () {}  
}
```

```
class MyString extends String{  
    public void print () {  
        System.out.println ("재정의");  
    }  
}
```



# 예외처리

- 예외란
- 예외 처리
- 사용자 정의 예외 생성

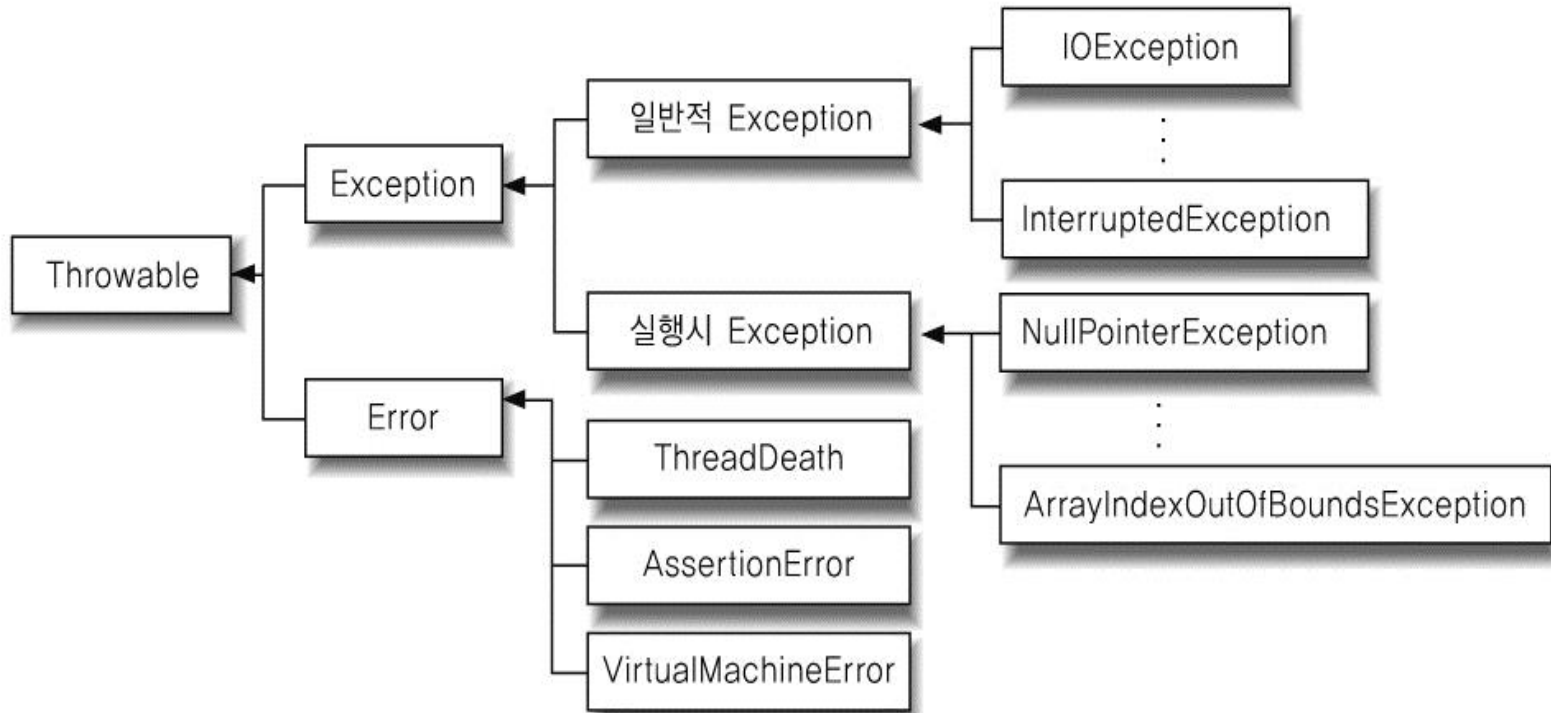


# 예외정의

---

- 예외 : 프로그램이 실행되는 동안에 발생하는 예기치 않은 에러
- 예외는 컴파일 시점과 실행 시점으로 나눌 수 있음
- 예외가 발생하는 예
  - 정수를 0으로 나누는 경우
  - 배열의 첨자가 음수 또는 범위를 벗어나는 경우
- 자바 언어는 프로그램에서 예외를 처리할 수 있는 기법을 제공
- 발생할 수 있는 예외를 클래스로 정의

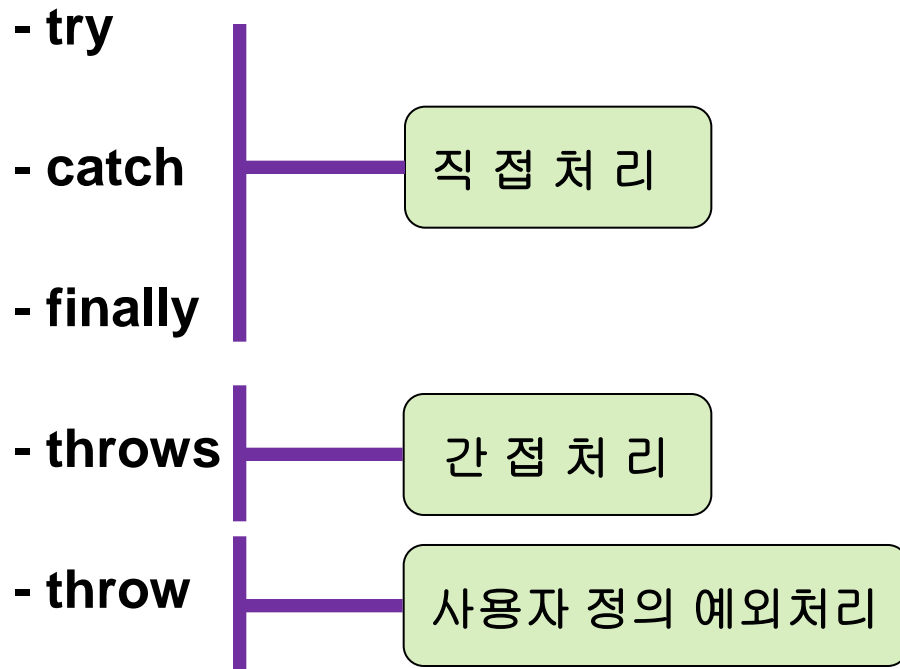
# 예외클래스



예외의 종류와 구조

# 예외관련 키워드

꼭 기억해야 할 키워드(5가지)





# 예외처리

## 1. 직접처리

- **try** : 예외가 발생할 만한 코드를 기술하는 부분
- **catch** : **try** 블록에서 예외가 발생하면 예외를 처리하는 부분
- **finally** : 예외 발생여부와 상관없이 무조건 실행하는 부분

```
try {  
    예외가 발생할 것 같은 코드 정의  
} catch(Exception e) {  
    예외처리  
} finally {           무조건 실행 }
```

### 예외처리 진행 순서

1. **try** 구문으로 진입

2 – 1. **try** 구문 안에서 예외가 발생하면

2 – 2. **catch** 구문을 순차적으로 살펴보면서 일치하는 예외가 있는지  
조사하여 해당 블록으로 간다.

2 – 3. 해당 **catch** 블록을 실행하여 예외처리를 한다.

2 – 2. **try** 구문 안에서 예외가 발생하지 않았다면 **catch** 블록을 실행하지  
않는다.

3. **finally** 블록이 있다면 예외 발생 유무에 상관없이 무조건 실행한다.

### 예외처리 진행 순서

1. try 구문으로 진입
- 2 - 1. try 구문 안에서 예외가 발생하면  
2 - 2. catch 구문을 순차적으로 실행하여  
조사하여 해당 블록으로 간다.  
2 - 3. 해당 catch 블록을 실행하여  
2 - 2. try 구문 안에서 예외가 발생했는지  
판단한다.
3. finally 블록이 있다면 예외 발생 유무

```
public static void main(String [] args) {  
    System.out.println(1);  
    try {  
        System.out.println(2);  
        예외 발생 가능 코드  
        System.out.println(3);  
    } catch(Exception e) {  
        System.out.println(4);  
    } finally {  
        System.out.println(5);  
    }  
}
```

정상 수행 시  
1 - 2 - 3 - 5

예외 발생 시  
1 - 2 - 4 - 5

**throws :**

메서드 내에서 발생한 예외를 자신이 직접 처리하는 것이 아니라  
자신을 호출한 쪽으로 예외처리를 떠넘기는 역할을 하는 키워드

형식 >

```
public void print( ) throws Exception {  
    예외가 발생할 것 같은 코드 정의  
}
```

**throw :**

**JVM**이 예외를 발생시키는 것이 아니라 인위적으로 특정 시점에 예외를 발생 시킬 때 사용하는 키워드

형식>

**throw** 예외객체

**throw new Exception( );**

사용자 정의 예외 클래스 :

정의 :

- **API**에 정의된 예외상황이 아니라 프로그램 내에서 특별한 예외 상황에 맞는 예외를 정의할 경우 사용

방법 :

- **Exception** 클래스를 상속받아서 정의

```
public class UserException extends Exception {  
    }  
}
```