

# 클래스, 객체

# 클래스객체기초

- 클래스(class) vs. 객체(object)

- new

- 필드변수 vs. 지역변수

- 객체 vs. 객체참조변수

# 클래스, 객체

## 클래스

- 대부분의 경우 객체 생성 및 객체에 대한 처리를 위한 코드
- 클래스는 자료형으로 사용될 수 있음

```
public class Date {  
    int year;  
    int month;  
    int day;  
}
```

필드(field)

```
public class Test {  
    public static void main(String[] args) {  
        Date date; ← Date 클래스의 객체 참조 변수 선언  
        date=new Date(); ← Date 객체 메모리 할당, 생성자 Date() 호출, 객체 메모리 참조 값 반환  
        date.year=2019; ← Date 객체의 year 필드에 2019 대입  
        date.month=3;  
        date.day=25;  
        System.out.println(date.year+"년"+date.month+"월"+date.day+"일");  
    }  
}
```

## 객체

- 클래스가 실체화된 것
- 클래스 관련 데이터 혹은 메소드의 모음

클래스 자료형

Date date = new Date() ;

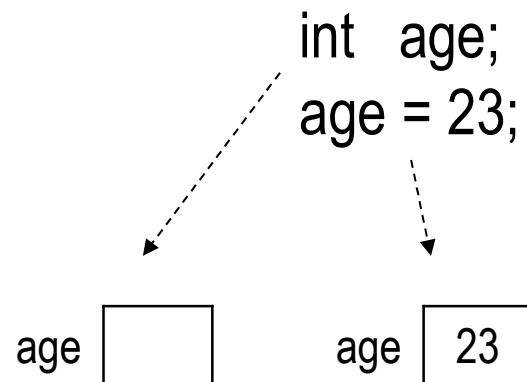
객체 참조 변수

- 클래스 Date의 객체 생성 (메모리 할당)
- 객체 생성자 Date() 호출
- 객체 메모리에 대한 참조 값 반환

# 기본 자료형 vs. 클래스 자료형

## 기본자료형

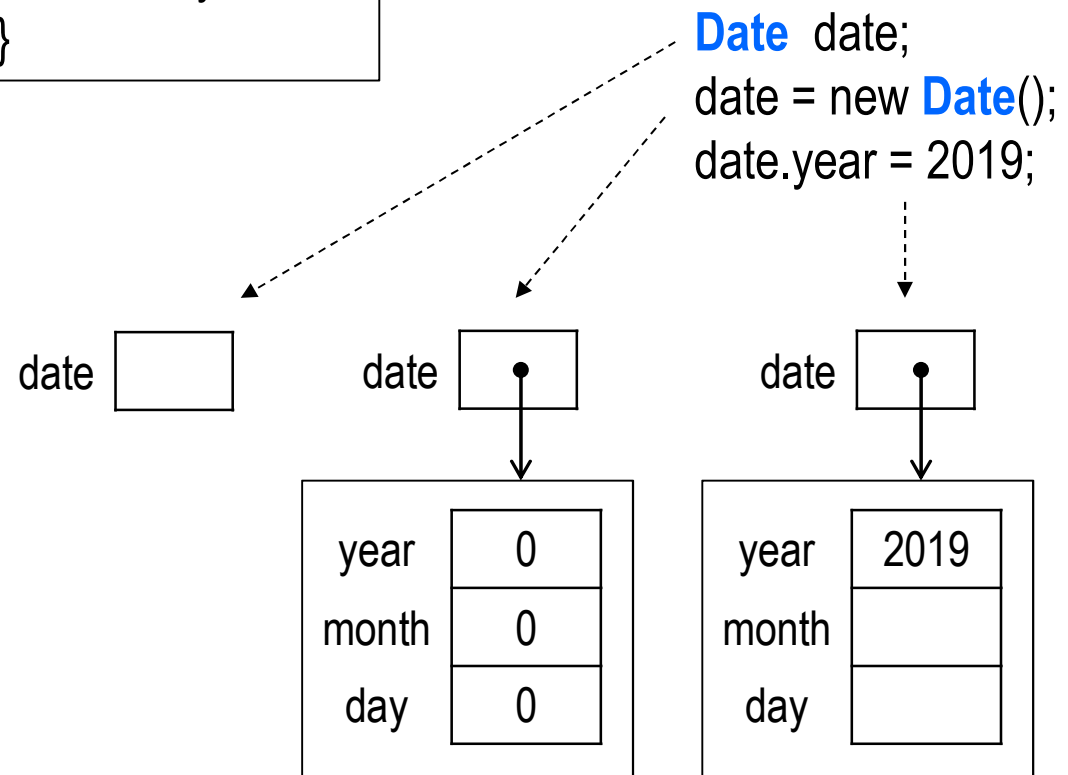
- 변수에 대응하는 메모리 공간에 데이터가 저장됨
- 변수 선언만으로 데이터를 저장할 메모리 공간 할당됨



```
public class Date {  
    int year;  
    int month;  
    int day;  
}
```

## 참조자료형(클래스자료형)

- 참조 변수에 대응하는 메모리 공간에 객체의 참조 값이 저장됨
- 변수 선언만으로 객체 공간 생성되지 않음. 별도 객체 생성 연산 수행 필요



# 객체 실습 A

- 다음은 Test 클래스의 미완성 코드와 그 실행 결과를 보인 것이다. 아래 코드가 정상 동작하도록 Student 클래스와 Test 클래스를 완성하시오.
  - ✓ Student 클래스의 필드 id, numberOfFinishedSemesters, gender, gpa, foreignerYN의 각각 문자열, 정수, 문자, 실수, 불리언 자료형으로 정의하시오.

```
public class Test {  
    public static void main(String[] args) {  
        Student s;  
  
        System.out.println("학 번="+s.id);  
        System.out.println("이수학기수="+s.numberOfFinishedSemesters);  
        System.out.println("성 별="+s.gender);  
        System.out.println("평 점="+s.gpa);  
        System.out.println("외국인 여부="+s.foreignerYN);  
    }  
}
```

## 실행결과

```
학 번=KSU-123  
이수학기수=2  
성 별=여  
평 점=3.97  
외국인 여부=true
```

# 필드, 배열 원소, 지역변수 초기값

참 조: <https://docs.oracle.com/javase/specs/jls/se8/html/jls-4.html#jls-4.12.5>

필드  
(field)

```
public class Player {  
    String    id;           // null  
    int       height;      // 0  
    double    record;      // 0.0d  
    boolean   dopingTestPostive; // false  
    char      gender;      // '\u0000'  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        int    score;  
        //System.out.println(score); // 지역변수  
  
        Player player=new Player();  
        System.out.println("id="+player.id);  
        System.out.println("신 장="+player.height);  
        System.out.println("기 록="+player.record);  
        System.out.println("도 핑 테스트 결과="+player.dopingTestPostive);  
        System.out.println("성 별="+player.gender);  
        System.out.printf("%d", (int)player.gender);  
    }  
}
```

## 필드(field), static 필드, 배열 원소

- 필드, static 필드 및 배열 원소는 객체 생성 시 아래 디폴트 값으로 자동 초기화됨
  - ✓ int → 0
  - ✓ double → 0.0
  - ✓ boolean → false
  - ✓ char → '\u0000'
  - ✓ 참조 변수 → null

## 지역변수

- 지역변수는 자동 초기화되지 않음
- 비초기화 지역변수는 사용 시 오류 발생

- field:** 필드, 객체 변수, instance variable
- static field:** 클래스 변수, class variable

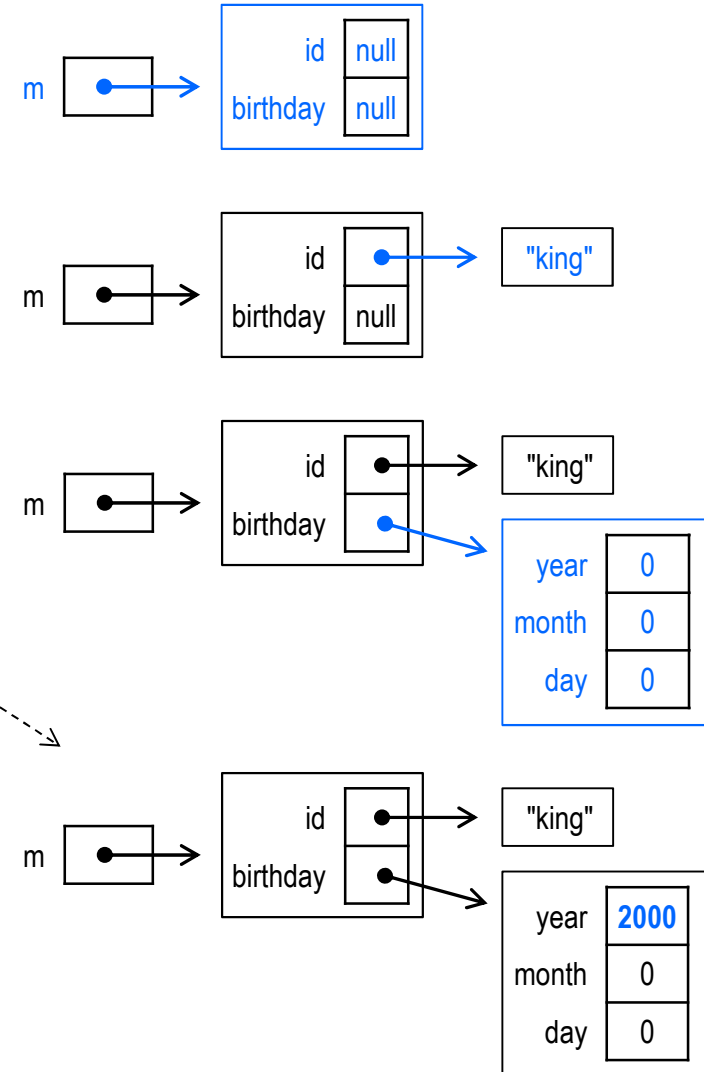
```
public class Test {  
    public static void main(String[] args) {  
        int n[]=new int[3]; // 배열 원소 초기화  
        System.out.println(n[0]); // 0  
    }  
}
```

# 클래스, 객체

```
public class Member {  
    String id;  
    YMD birthday;  
}
```

```
public class YMD {  
    int year;  
    int month;  
    int day;  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Member m=new Member();  
        m.id = "king";  
        m.birthday=new YMD();  
        m.birthday.year=2000;  
        m.birthday.month=12;  
        m.birthday.day=31;  
        System.out.println("회원아이디=" + m.id);  
        System.out.println("회원출생년=" + m.birthday.year);  
        System.out.println("회원출생월=" + m.birthday.month);  
        System.out.println("회원출생일=" + m.birthday.day);  
    }  
}
```



# 객체 vs. 객체 참조 변수

```
public class YMD {  
    int    year;  
    int    month;  
    int    day;  
}
```

```
public class Test {  
    public static void main(String[] args) {
```

```
        YMD date1=null;  
        date1=new YMD();  
        date1.year=2020;  
        date1.month=4;  
        date1.day=1;
```

```
        YMD date2=null;  
        date2=date1;
```

```
        YMD date3=date1;
```

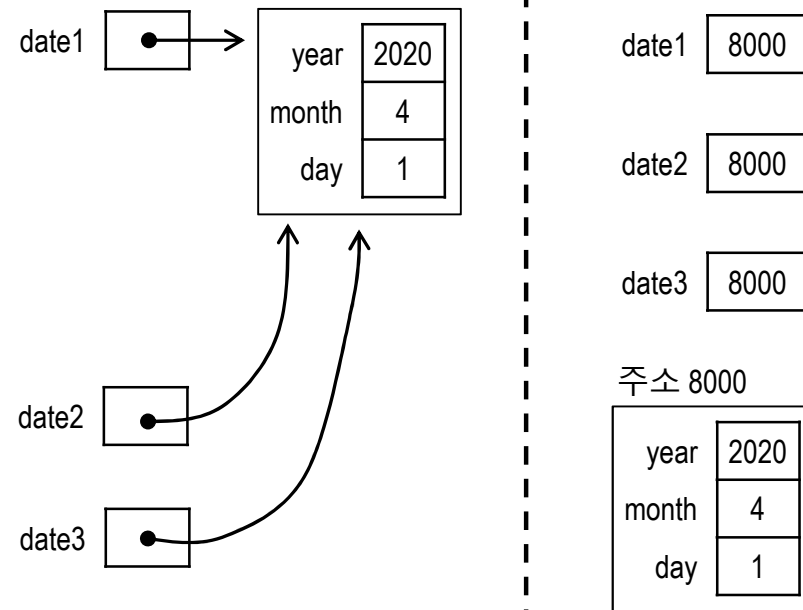
```
        System.out.println(date1.year+"년"+date1.month+"월"+date1.day+"일");  
        System.out.println(date2.year+"년"+date2.month+"월"+date2.day+"일");  
        System.out.println(date3.year+"년"+date3.month+"월"+date3.day+"일");
```

```
    }  
}
```

**date2 = date1;**

- date1에 저장된  
 객체참조값(예: 8000)을  
 date2에 대입 저장

객체참조변수에는 객체에 대한  
참조값(주소값)이 저장됨





## 객체 실습 B

- 날짜 클래스 DateInfo를 다음과 같이 정의하시오
  - 필드변수명(설명,자료형): year(연도,int), month(월,int), day(일,int)
- 직원 클래스 Employee를 다음과 같이 정의하시오
  - 필드변수명(설명,자료형): id(직원아이디,String), joinDate(직원입사일,DateInfo)
- 클래스 Test의 main 메소드 내에 아래 절차를 코딩하시오
  - 다음 두 직원의 정보를 각각 Employee 객체로 만드시오(객체참조변수명 e1, e2 사용)
    - 직원 1 → 직원아이디 = EMP-123, 직원입사일 = 2015년3월2일
    - 직원 2 → 직원아이디 = EMP-456, 직원입사일 = 2016년3월4일
  - 위 두 객체의 참조값을 크기 2의 배열 v에 저장하시오
  - for문을 사용하여 배열 v 내 각 객체의 정보를 아래와 같이 출력하시오

직원번호: EMP-123, 입사일: 2015년3월2일 직원번호: EMP-456, 입사일: 2016년3월4일
--

# 클래스, 객체

생성자, this, toString()

# 생성자(Constructor)

생성자

- 클래스 이름과 동일한 이름
- 반환 자료형 표기 없음

**Student.java**

```
public class Student {  
    String    name;  
    int       score;  
}
```

**Student.java**

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String _name, int _score) {  
        name=_name;  
        score=_score;  
    }  
}
```



**Test.java**

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student();  
        student.name="김철수";  
        student.score=89;  
        System.out.println(student.name);  
        System.out.println(student.score);  
    }  
}
```

**Test.java**

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student("김철수", 89);  
        System.out.println(student.name);  
        System.out.println(student.score);  
    }  
}
```



# this

## *this*

- 현재 객체에 대한 참조값이 저장된 변수
- 생성자, instance method 등에서 사용

### Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String _name, int _score) {  
        name=_name;  
        score=_score;  
    }  
}
```



### Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
}
```

# 기본생성자(default constructor)

## Student.java

```
public class Student {  
    String    name;  
    int       score;  
}
```

## Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student();  
        student.name="김철수";  
        student.score=89;  
        System.out.println(student.name);  
        System.out.println(student.score);  
    }  
}
```

## Student.java

```
public class Student {  
    String name;  
    int score;  
  
    public Student() {  
    }  
}
```

### 생성자

- 클래스 이름과 동일한 이름을 가짐
- 반환 자료형 표기 없음
- 기본생성자(default constructor)는 파라미터 없는 생성자
- 클래스 정의문 내 생성자가 없으면 컴파일러가 자동으로 기본생성자를 생성

# 기본생성자(default constructor)

## Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
}
```

## Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student(); ←  
    }  
}
```

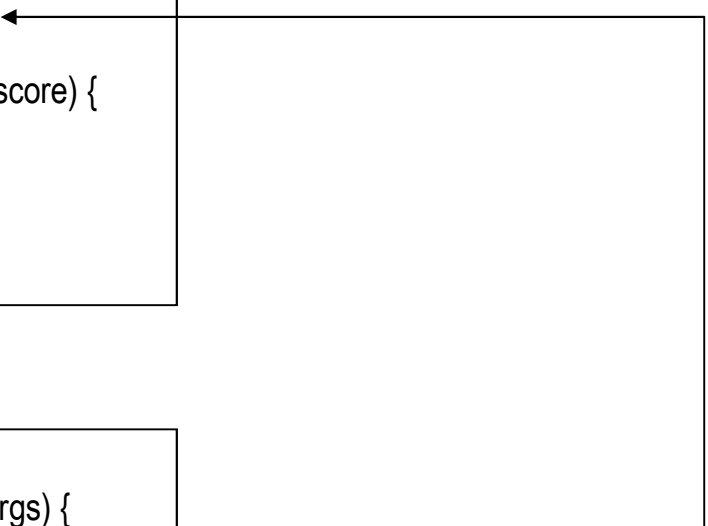
오류

- Student 클래스에 생성자가 존재하므로 기본생성자 Student()는 생성되지 않음
- 즉 기본생성자 Student()는 존재하지 않음

# 기본생성자(default constructor)

## Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student() {  
    }  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
}
```



## Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student();  
    }  
}
```

- 다른 생성자가 있는 경우 기본생성자는 자동 생성되지 않음
- 다른 생성자가 정의되어 있지만 Test.java의 예와 같이 기본생성자 사용도 필요하다면 Student 클래스 정의문에 기본생성자 정의를 추가해야 함

# 클래스 실습 A

- 다음 조건에 따라 클래스 Car를 정의하시오
  - 필드변수명(설명,자료형): id(고유번호,String), length(길이,int), weight(중량,double), fuelType(연료유형,char), exportOnly(수출용여부,boolean)
  - 고유번호, 길이, 중량, 연료유형, 수출용여부 값을 파라미터로 전달받아 대응하는 필드에 대입하는 생성자
- 다음은 Car 객체를 생성하고 그 내용을 출력하는 코드와 실행 결과를 보인 것이다. 이 코드가 정상 동작하도록 Car 클래스를 정의하시오.

```
public class Test {  
    public static void main(String[] args) {  
        Car car=new Car("KSC-123", 3850, 1500.8, 'D', false);  
        System.out.println("고유번호="+car.id);  
        System.out.println("길이(mm)="+car.length);  
        System.out.println("중량(kg)="+car.weight);  
        System.out.println("연료유형="+car.fuelType);  
        System.out.println("수출용="+car.exportOnly);  
    }  
}
```

실행결과

```
고유번호=KSC-123  
길이(mm)=3850  
중량(kg)=1500.8  
연료유형=D  
수출용=false
```



# toString()

## toString()

- 객체의 정보들을 하나의 문자열로 만들어 반환하는 메소드 (객체에 대한 문자열 표현을 String으로 반환)

### Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
}
```



### Student.java

```
public class Student {  
    String    name;  
    int       score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
    @Override  
    public String toString() {  
        return name+","+score;  
    }  
}
```

### Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student("이영희", 89);  
        System.out.println(student.name+" "+student.score);  
    }  
}
```



### Test.java

```
public class Test {  
    public static void main(String[] args) {  
        Student student;  
        student=new Student("이영희", 89);  
        System.out.println(student.toString());  
        System.out.println(student);  
    }  
}
```

# 클래스 실습 B

- 다음 조건에 따라 클래스 YMD를 정의하시오
  - 필드변수명(설명,자료형): year(연도,int), month(월, int), day(일, int)
  - 연,월,일을 파라미터로 전달받아 대응하는 필드에 대입하는 생성자
  - 연,월,일 값을 다음 예와 같은 형식의 문자열로 반환하는 toString() 메소드 (예: **2019년12월31일**)
- 다음은 YMD 객체를 생성하고 그 내용을 출력하는 코드와 실행 결과를 보인 것이다. 이 코드가 정상 동작하도록 YMD 클래스를 정의하시오.

```
public class Test {  
    public static void main(String[] args) {  
        YMD date=new YMD(2019, 12, 31);  
        System.out.println(date.toString());  
        System.out.println(date);  
    }  
}
```

실행결과

```
2019년12월31일  
2019년12월31일
```

# 클래스, 객체

객체 메소드, private, 오버로딩, this(), static

# 객체 메소드: getter, setter

```
public class Student {  
    String name;  
    int    score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
    @Override  
    public String toString() {  
        return "이름:"+name+", 점수:"+score;  
    }  
    public String getName() {  
        return name;  
    }  
    public int getScore() {  
        return score;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void setScore(int score) {  
        this.score = score;  
    }  
}
```

## 객체 메소드

- 객체에 적용될 절차를 함수 형식으로 해당 객체의 클래스 내부에 기술한 코드

## 객체 메소드 예

- **getter** 메소드: 객체의 필드 값을 반환하는 메소드  
✓ getName(), getScore()
- **setter** 메소드: 객체의 필드 값을 변경하는 메소드  
✓ setName(), setScore()

```
public class Test {  
    public static void main(String[] args) {  
        Student    s=new Student("이영희", 95);  
        System.out.println(s);  
  
        String name=s.getName();  
        int    score=s.getScore();  
        s.setName(name+"(오)");  
        s.setScore(score+2);  
  
        System.out.println(s);  
    }  
}
```

## 객체 메소드 실습 A

```
public class Player {  
    String    id;  
    int    record1, record2, record3;  
    public Player(String id, int record1, int record2, int record3) {  
        this.id=id;  
        this.record1=record1;  
        this.record2=record2;  
        this.record3=record3;  
    }  
}
```

**실습:** Player 클래스에 객체 메소드 `getAverage()`를 정의하여 아래 코드가 p에 저장된 선수의 평균을 출력하도록 하시오.

```
public class Test {  
    public static void main(String[] args) {  
        Player p=new Player("P001", 210, 205, 220);  
        System.out.println(p.getAverage());  
    }  
}
```

# 접근 제어자 (access modifier)

참조: <https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>

## 접근 제어자

- 한 클래스의 필드 및 메소드를 다른 클래스에서 접근(사용하거나 호출) 가능하게 할지 여부를 결정하는 키워드

접근 제어자	클래스	패키지	하위클래스	외부
public	O	O	O	O
protected	O	O	O	X
no modifier	O	O	X	X
private	O	X	X	X

- private → 클래스 내부에서만 접근 가능, 클래스 외부에서는 접근 불가
- no modifier (default) → 클래스 외부더라도 같은 패키지 내 클래스에서까지 접근 가능
- protected → 패키지 외부더라도 하위 클래스에서까지 접근 가능
- public → 클래스 내부 및 외부 어디에서든 접근 가능

# 정보은닉: private, getter, setter

- ✚ 정보은닉을 보장하는 수단으로 private, getter, setter 사용
  - private → 다른 모든 클래스로부터의 접근 불허 (자신 클래스 내부로부터의 접근만 허용)
  - getter → 객체의 속성 값을 반환하는 메소드
  - setter → 객체의 속성 값을 변경하는 메소드

```
public class Member {  
    private String id;  
    private String name;  
    public Member(String id, String name) {  
        this.id=id;  
        this.name=name;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    @Override  
    public String toString() {  
        return id+","+name;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Member member;  
        member=new Member("M-0123", "이영희");  
        member.setName("김철수");  
        System.out.println(member.getName());  
    }  
}
```

# private 실습 A

✚ 회원 클래스 Member를 다음 조건에 따라 작성하시오

- 필드 변수명(설명, 자료형): email(메일주소, String), joinDate(가입일-8자리, String)
  - ◆ 클래스 외부로부터의 직접 접근 불허 설정할 것
- 메일주소와 가입일을 파라미터로 전달받아 대응하는 필드에 대입하는 생성자
- 회원의 email을 반환하는 public 메소드 getEmail()
- 새로운 메일주소를 파라미터로 전달받아 객체의 기존 메일주소를 변경하는 메소드 setEmail()
- 회원의 joinDate를 8자리 숫자 문자열로 반환하는 public 메소드 getJoinDate()
- 회원의 가입년도를 4자리 숫자 문자열로 반환하는 public 메소드 getJoinYear()

✚ 클래스 Test의 main() 내 다음 절차를 코딩하시오

- Member 객체(메일주소 yhkim@ks.ac.kr, 가입일 20190214 ) 생성
- Member 객체의 메일주소와 가입일을 출력
- Member 객체의 메일주소를 yhkim@cs.ks.ac.kr로 변경
- Member 객체의 메일주소와 가입일을 출력
- Member 객체의 가입년도를 출력



# 오버로딩(overloading)

## 오버로딩(overloading)

- 파라미터의 개수나 타입이 다르면서 이름이 같은 생성자 혹은 메소드를 정의하는 것 (리턴 타입만 다른 경우는 오버로딩 아님, 오류 발생)

```
public class Member {  
    String    id;  
    char      gender;  
    public Member(String id) {  
        this.id=id;  
    }  
    public Member(String id, char gender) {  
        this.id=id;  
        this.gender=gender;  
    }  
    @Override  
    public String toString() {  
        return id+","+gender;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Member    member1, member2;  
        member1=new Member("M-123");  
        member2=new Member("M-124",'여');  
        System.out.println(member1);  
        System.out.println(member2);  
    }  
}
```

# 오버로딩(overloading)

```
public class Customer {
    String email;
    double point;
    public Customer(String email) {
        this.email=email;
    }
    public void addToPoint(int point) {
        this.point+=point;
    }
    public void addToPoint(double point) {
        this.point+=point;
    }
    public void addToPoint(int p1, int p2) {
        this.point+=p1+p2;
    }
    public void addToPoint(int[] v) {
        for (int i = 0; i < v.length; i++) {
            this.point+=v[i];
        }
    }
    @Override
    public String toString() {
        return email+","+point;
    }
}
```



## 오버로딩(overloading)

- 파라미터의 개수나 타입이 다르면서 이름이 같은 생성자 혹은 메소드를 정의하는 것 (리턴 타입만 다른 경우는 오버로딩 아님, 오류 발생)

```
public class Test {
    public static void main(String[] args) {
        Customer c=new Customer("goodmorning@ks.ac.kr");
        c.addToPoint(10);
        c.addToPoint(45.3);
        c.addToPoint(10, 20);
        int v[]={2,7,1};
        c.addToPoint(v);
        System.out.println(c);
    }
}
```

# 오버로딩(overloading)

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println(100);  
        System.out.println(3.14);  
        System.out.println('봄');  
        System.out.println(true);  
        System.out.println("대한민국");  
    }  
}
```

# 클래스 작성: this()

## this()

- 한 클래스 내의 다른 생성자를 호출할 때 사용
  - ◆ 아래 코드에서 this(id, name);을 Member(id, name);로 작성하면 오류 발생

```
public class Member {  
    String    id;  
    String    name;  
    char      gender;  
    public Member(String id, String name) {  
        this.id=id;  
        this.name=name;  
    }  
    public Member(String id, String name, char gender) {  
        this.id=id;  
        this.name=name;  
        this.gender=gender;  
    }  
    @Override  
    public String toString() {  
        return id+","+name+","+gender;  
    }  
}
```



```
public class Member {  
    String    id;  
    String    name;  
    char      gender;  
    public Member(String id, String name) {  
        this.id=id;  
        this.name=name;  
    }  
    public Member(String id, String name, char gender) {  
        this(id, name);  
        this.gender=gender;  
    }  
    @Override  
    public String toString() {  
        return id+","+name+","+gender;  
    }  
}
```

다른 생성자 호출문은 생성자 내 첫 행에 위치  
참조: <https://docs.oracle.com/javase/tutorial/java/javaOO/thiskey.html>

# 오버로딩 실습 A

```
public class Employee {  
    String id;  
    double monthlyPayBase=250; // 기본 월 급여액 250만원  
    public Employee(String id) {  
        this.id=id;  
    }  
    public double getMonthlyPay() {  
        return monthlyPayBase;  
    }  
}
```

## 실습

- 아래 Test 클래스가 정상 동작하도록 Employee 클래스를 수정하시오
- 생성자 오버로딩 시 this()를 사용하시오

```
public class Test {  
    public static void main(String[] args) {  
        Employee e1=new Employee("E-001"); // 일반 직원  
        Employee e2=new Employee("E-002", 100); // 인턴 직원, 기본 월 급여액 100만원  
        System.out.println(e1.getMonthlyPay()); // 월 급여액 출력  
        System.out.println(e2.getMonthlyPay()); // 월 급여액 출력  
    }  
}
```

# 인스턴스 변수, 인스턴스 메소드

- 인스턴스 변수(instance variable), (non-static) 필드
  - 특정 객체 고유의 값을 보관, 객체 생성 이후 사용 가능
- 인스턴스 메소드 (instance method), 객체 메소드
  - 특정 객체에 적용되는 메소드, 객체 생성 이후 사용 가능
- 인스턴스 변수 및 인스턴스 메소드는 객체 생성 후 객체 참조 값을 통해 접근

```
public class Robot {  
    String id; // 필드, 인스턴스 변수  
    public Robot(String id) {  
        this.id=id;  
    }  
    // 인스턴스 메소드, 객체 메소드  
    public void changeld(String id) {  
        this.id=id;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Robot robot1=new Robot("R001");  
        robot1.id="R100";           // 객체 필드 값 변경  
        robot1.changeld("R111");    // 인스턴스 메소드 호출  
        // Robot.id="R200";         // 오류  
        // Robot.changeld("R222");  // 오류  
    }  
}
```

# 클래스 변수, 클래스 메소드

- 클래스 변수(class variable, static field)
  - 클래스 관련 데이터를 보관, 객체 생성 이전 사용 가능, 변수 선언 시 **static 키워드** 부착
- 클래스 메소드 (class method, static method)
  - 클래스 수준의 절차를 명시, 객체 생성 이전 사용 가능, 메소드 정의 시 **static 키워드** 부착
- 클래스 변수 및 클래스 메소드는 객체 참조 값이 아닌 클래스명을 통해 접근
  - 클래스 변수 및 클래스 메소드는 객체 참조 값을 통해서도 접근 가능하나 이는 가독성 측면에서 지양

```
public class Robot {  
    String id;  
    static int numRobot; // 클래스 변수, static 필드  
    public Robot(String id) {  
        this.id=id;  
        numRobot++;  
    }  
    public void changeId(String id) {  
        this.id=id;  
    }  
    public static int getNumRobot() { // 클래스 메소드  
        return numRobot;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Robot robot1=new Robot("R001");  
        Robot robot2=new Robot("R002");  
  
        System.out.println(Robot.numRobot);  
        // System.out.println(robot1.numRobot);  
        // System.out.println(robot2.numRobot);  
  
        System.out.println(Robot.getNumRobot());  
        // System.out.println(robot1.getNumRobot());  
        // System.out.println(robot2.getNumRobot());  
    }  
}
```

# static 필드 (static field)

Non-static 필드  
인스턴스(instance) 변수

```
public class Employee {  
    String id;  
    double monthlyPayBase=250; // 월기본급  
    public Employee(String id) {  
        this.id=id;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Employee e1=new Employee("E-001");  
        Employee e2=new Employee("E-002");  
        System.out.println(e1.monthlyPayBase);  
        System.out.println(e2.monthlyPayBase);  
    }  
}
```

- 클래스 Employee의 monthlyPayBase는 객체 공통 데이터임 (이 회사의 월기본급은 직원마다 다르지 않다고 가정)
- 이 경우 개별 직원 객체를 통해 월기본급을 확인하는 것은 비효율적

static 필드  
클래스 변수

```
public class Employee {  
    String id;  
    static double monthlyPayBase=250;  
    public Employee(String id) {  
        this.id=id;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        System.out.println(Employee.monthlyPayBase);  
    }  
}
```

- 클래스 Employee의 monthlyPayBase를 static 필드로 정의
- static 필드는 객체를 생성하지 않고 클래스를 통해 접근 가능



# static 메소드 (static method)

Non-static 메소드  
인스턴스(instance) 메소드

```
public class Calculator {  
    public int sum(int x, int y) {  
        return x+y;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Calculator c1=new Calculator();  
        int v1=c1.sum(11,22);  
        System.out.println(v1);  
        Calculator c2=new Calculator();  
        int v2=c2.sum(33,44);  
        System.out.println(v2);  
    }  
}
```

- 클래스 Calculator에는 객체 고유의 자료 (서로 다른 객체마다 서로 다른 자료)를 관리하지 않음
- 이 경우 메소드 sum()을 인스턴스 메소드로 정의하는 것은 비효율적 (sum() 호출을 위해 Calculator 객체를 생성해야 함)

static 메소드  
클래스 메소드

```
public class Calculator {  
    public static int sum(int x, int y) {  
        return x+y;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        int v1=Calculator.sum(11,22);  
        System.out.println(v1);  
  
        int v2=Calculator.sum(33,44);  
        System.out.println(v2);  
    }  
}
```

- 클래스 Calculator의 sum() 메소드를 static 메소드로 정의 (static 키워드 부착)
- static 메소드는 객체를 생성하지 않고 클래스를 통해 호출 가능

## static 필드, static 메소드 사용 예

```
public class Test {  
    public static void main(String[] args) {  
        int  max=Math.max(43, 76);  
        System.out.println(max); // 76  
        int  v=Integer.parseInt("123")+5;  
        System.out.println(v); // 128  
        double  w=Double.parseDouble("3.14")+0.05;  
        System.out.println(w); // 3.19  
        long  curTime=System.currentTimeMillis();  
        System.out.println(curTime);  
        System.out.println(Math.PI); // 원주율  
        System.out.println(Color.RED);  
        double  x=Math.pow(2, 10); // 2의 10승  
        System.out.println(x); // 1024.0  
    }  
}
```

## static 실습 A

- 다음 Test 클래스가 정상 동작하도록 클래스 Calc를 작성하시오
- Calc.max()는 파라미터로 전달받은 두 정수 중 큰 값을 반환한다
- Calc.areaOfCircle()은 원의 반지름을 파라미터로 전달받아 원의 면적을 반환한다
- 원의 면적 계산을 위해 static field 변수 PI에 저장된 원주율값(3.14159로 가정)을 사용하시오

```
public class Test {  
    public static void main(String[] args) {  
        int max=Calc.max(12,34);  
        System.out.println(max);  
        System.out.println("원주율="+Calc.PI);  
        double radius=2.0;  
        double area=Calc.areaOfCircle(radius);  
        System.out.println("원의 면적="+area);  
    }  
}
```

# 클래스, 객체

상속, Object, super, super(), 오버라이딩, 상속관계 객체참조, type casting, instanceof

# 클래스 작성: 상속

다음 두 클래스의 정의를 비교하시오

```
public class Robot{  
    String id;  
    public Robot(String id) {  
        this.id=id;  
    }  
    @Override  
    public String toString() {  
        return id;  
    }  
}
```

```
public class HumanoidRobot{  
    String id;      // 제조회사에서 부여된 식별번호  
    String address; // 로봇의 주소  
    String name;    // 로봇에게 부여된 이름  
    public HumanoidRobot(String id, String address, String name){  
        this.id=id;  
        this.address=address;  
        this.name=name;  
    }  
    @Override  
    public String toString() {  
        return id+","+address+","+name;  
    }  
}
```

# 클래스 작성: 상속

## 클래스 상속(inheritance)

- 이미 작성된 Robot 클래스가 존재한다고 가정할 때
- Robot 클래스의 정의를 재사용(확장, extends)하여 HumanoidRobot 클래스를 정의

```
public class Robot{  
    String id;  
    public Robot(String id) {  
        this.id=id;  
    }  
    @Override  
    public String toString() {  
        return id;  
    }  
}
```

```
public class HumanoidRobot extends Robot{  
    String address;  
    String name;  
    public HumanoidRobot(String id, String address, String name) {  
        super(id);  
        this.address=address;  
        this.name=name;  
    }  
    @Override  
    public String toString() {  
        return super.toString()+","+address+","+name;  
    }  
}
```

# 오버라이딩 vs. 오버로딩

## 오버라이딩(overriding)

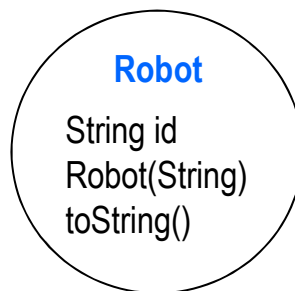
- 상위 클래스 내 메소드와 파라미터 개수, 파라미터 타입 및 리턴 타입이 모두 같은 메소드를 하위 클래스 내에 정의하는 것

## 오버로딩(overloading)

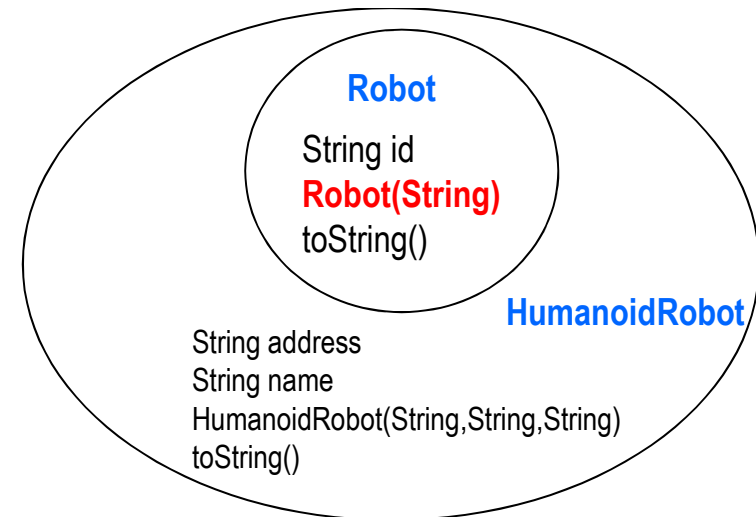
- 파라미터의 개수나 타입이 다르면서 이름이 같은 생성자 혹은 메소드를 정의하는 것 (리턴 타입만 다른 경우는 오버로딩 아님)

# 클래스 작성: 상속과 포함관계

```
public class Robot {  
    String id;  
    public Robot(String id) {  
        this.id=id;  
    }  
    @Override  
    public String toString() {  
        return id;  
    }  
}
```



```
public class HumanoidRobot extends Robot {  
    String address;  
    String name;  
    public HumanoidRobot(String id, String address, String name) {  
        super(id);  
        this.address=address;  
        this.name=name;  
    }  
    @Override  
    public String toString() {  
        return super.toString()+","+address+","+name;  
    }  
}
```



출처: <https://docs.oracle.com/javase/tutorial/java/landl/subclasses.html>

- 하위클래스는 상위클래스의 접근 가능한 모든 member들을 상속
- 생성자는 member가 아니므로 상속되는 것은 아니며 하위클래스로부터 **super()**를 통해 호출 가능



# 클래스 Object

## 클래스 Object

- Object를 제외한 다른 모든 자바 클래스의 부모 혹은 조상 클래스
- 클래스 Object는 부모 클래스가 없음
- 부모 클래스 없이 정의된 클래스는 암묵적으로 Object의 자식 클래스

```
public class Person {  
  
}
```

```
public class Person extends Object {  
  
}
```

# 상속 클래스 정의: super()

```
public class Person {  
    String name;  
    public Person() {  
        System.out.println("Person 객체 생성");  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student() {  
        System.out.println("Student 객체 생성");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student();  
    }  
}
```

실행결과:  
Person 객체 생성  
Student 객체 생성

```
public class Person {  
    String name;  
    public Person() {  
        System.out.println("Person 객체 생성");  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student() {  
        super();  
        System.out.println("Student 객체 생성");  
    }  
}
```

부모클래스 생성자 호출문 누락 시  
부모클래스 기본생성자 호출문  
super(); 자동 삽입

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student();  
    }  
}
```

```
public class Person extends Object {  
    String name;  
    public Person() {  
        super();  
        System.out.println("Person 객체 생성");  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student() {  
        super();  
        System.out.println("Student 객체 생성");  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student();  
    }  
}
```

# 상속 클래스 정의: super()

```
public class Person {  
    String name;  
    public Person(String name) {  
        this.name=name;  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student(String name, int score) {  
        this.name=name;  
        this.score=score;  
    }  
}
```

- 오류 발생

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student("홍길동", 98);  
    }  
}
```

```
public class Person {  
    String name;  
    public Person(String name) {  
        this.name=name;  
    }  
}
```

Person 클래스 내 생성자  
존재하므로 기본 생성자  
자동 삽입되지 않음

```
public class Student extends Person {  
    int score;  
    public Student(String name, int score) {  
        super();  
        this.name=name;  
        this.score=score;  
    }  
}
```

- 오류 발생  
부모클래스 생성자 호출문 누락 시  
부모클래스 기본생성자 호출문 super();  
자동 삽입되나, 부모클래스 Person에는  
기본 생성자 부재

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student("홍길동", 98);  
    }  
}
```

```
public class Person {  
    String name;  
    public Person(String name) {  
        this.name=name;  
    }  
    public Person() {  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student(String name, int score) {  
        super();  
        this.name=name;  
        this.score=score;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Students=new Student("홍길동", 98);  
    }  
}
```

## 상속 클래스 정의: super()

```
public class Person {  
}
```

```
public class Student extends Person {  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s=new Student();  
    }  
}
```

```
public class Person {  
    public Person() {  
        super();  
    }  
}
```

```
public class Student extends Person {  
    public Student() {  
        super();  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s=new Student();  
    }  
}
```

# 상속 실습 A



다음 조건을 만족하는 클래스 Book을 작성

- 필드변수명(설명,자료형): id (도서고유번호, String), title (서명, String), year (출판년도, int)
- id, title, year 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자 작성
- id, title, year 값을 다음 형식으로 반환하는 toString() 작성
  - ◆ 예) B00012, 자바 시작하기, 2018(년)



다음 조건을 만족하는 클래스 EBook을 작성

- 클래스 Book을 상속
- 필드변수명(설명,자료형): fileSize (파일크기, double)
- id, title, year, fileSize를 파라미터로 전달받아 대응하는 필드에 저장하는 생성자 작성
  - ◆ Book 클래스의 생성자 호출할 것
- id, title, year, fileSize 값을 다음 형식으로 반환하는 toString() 작성. Book 클래스의 toString() 호출할 것
  - ◆ 예) B00012, 자바 시작하기, 2018(년), 20.4(MB)



클래스 Test의 main 메소드 내 다음 절차 수행 코드 작성

- 다음 표에 대응하는 EBook 객체 생성

id	title	year	fileSize (MB)
B00012	자바 시작하기	2018	20.4

- 위 EBook 객체 정보 출력

# 상속 관계 객체 참조

```
public class Person {  
    String name;  
    public Person(String name) {  
        this.name=name;  
    }  
    @Override  
    public String toString() {  
        return "사람";  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student(String name, int score) {  
        super(name);  
        this.score=score;  
    }  
    @Override  
    public String toString() {  
        return "학생";  
    }  
}
```

```
public class Teacher extends Person {  
    String subject;  
    public Teacher(String name, String subject) {  
        super(name);  
        this.subject=subject;  
    }  
    @Override  
    public String toString() {  
        return "선생님";  
    }  
}
```

Person  
|  
Student

Student is-a Person (O)  
Person is-a Student (X)

```
Student s=new Person("이영희");
```

← 자식클래스 참조변수로  
부모클래스 객체 참조 불가

```
Person p=new Student("홍길동", 98);
```

← 부모클래스 참조변수로  
자식클래스 객체 참조 가능

# 상속 관계 객체 참조

```
public class Person {  
    String name;  
    public Person(String name) {  
        this.name=name;  
    }  
    @Override  
    public String toString() {  
        return name;  
    }  
}
```

```
public class Student extends Person {  
    int score;  
    public Student(String name, int score) {  
        super(name);  
        this.score=score;  
    }  
    @Override  
    public String toString() {  
        return "학생: "+super.toString()+" "+score;  
    }  
}
```

```
public class Teacher extends Person {  
    String subject;  
    public Teacher(String name, String subject) {  
        super(name);  
        this.subject=subject;  
    }  
    @Override  
    public String toString() {  
        return "선생님: "+super.toString()+" "+subject;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s=new Student("김철수", 90);  
        Teacher t=new Teacher("이영희", "수학");  
        Person persons[]={s, t};  
        Object objects[]={s, t};  
        println(persons);  
        println(objects);  
    }  
    private static void println(Object[] objects) {  
        for (Object o : objects) System.out.println(o);  
    }  
    private static void println(Person[] persons) {  
        for (Person p : persons) System.out.println(p);  
    }  
}
```

## 실행결과

학생: 김철수,90

선생님: 이영희,수학

학생: 김철수,90

선생님: 이영희,수학

## 상속 실습 B

- ✚ 클래스 Shape(도형)을 다음 조건에 따라 정의하시오
  - 실수 0.0을 반환하는 메소드 `getArea()`
- ✚ 클래스 Rectangle(사각형)을 클래스 Shape을 상속받아 다음 조건에 따라 정의하시오
  - 필드: `width`(가로 길이, 정수), `height`(세로 길이, 정수)
  - `width`, `height` 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
  - 사각형의 면적을 실수로 반환하는 메소드 `getArea()`을 Shape의 `getArea()`를 오버라이드하여 작성
- ✚ 클래스 Circle(원)을 클래스 Shape을 상속받아 다음 조건에 따라 정의하시오
  - 필드: `radius`(반지름, 정수)
  - `radius` 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
  - 원의 면적을 실수로 반환하는 메소드 `getArea()`을 Shape의 `getArea()`를 오버라이드하여 작성
- ✚ 클래스 Test의 `main()` 내에 다음 절차를 코딩하시오
  - 다음 객체들을 **Shape 배열**에 저장
    - ◆ Rectangle 객체(가로 3, 세로 4), Circle 객체(반지름 5), Circle 객체(반지름 2)
  - 반복문을 통해 배열에 저장된 각 도형의 면적(`getArea()` 호출)을 출력
  - 반복문을 통해 배열에 저장된 각 도형의 면적(`getArea()` 호출)을 도형 유형(사각형, 원)과 함께 출력



# 클래스, 객체

추상클래스, 추상메소드, 인터페이스, 내포클래스,  
익명내부클래스, ActionListener

# 클래스 작성: 추상클래스, 추상메소드



## 실체 클래스 vs. 추상클래스

- 실체클래스(Concrete class)
  - ◆ 객체화 가능한 클래스
- 추상클래스(Abstract class)
  - ◆ Abstract 키워드가 부착된 클래스로 객체화될 수 없는 클래스
  - ◆ 추상메소드를 포함할 수 있음
    - 추상메소드는 그 구현 부분이 누락된 메소드 (아래 setColor() 참조)
      - 메소드 구현이 기술되는 { ... } 부분이 없으며 마지막에 세미콜론으로 종료
  - ◆ 자식 클래스로의 상속은 가능

```
public class Person {  
    String id;  
    String name;  
    public Person(String id, String name) {  
        this.id=id;  
        this.name=name;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Person person=new Person("S123", "Obama");  
    }  
}  
new 연산자를 통해 Person 객체 생성 가능
```

```
public abstract class Colorable {  
    abstract void setColor(int color);  
}
```

추상클래스  
(abstract class)

추상메소드  
(abstract method)

```
public class Test {  
    public static void main(String[] args) {  
        Colorable colorable=new Colorable(); // 오류 !  
    }  
}  
new 연산자를 통해 Colorable 객체 생성 불가  
setColor() 메소드 구현 부분이 존재하지 않음
```

# 클래스 작성: 추상클래스, 인터페이스, final



## 추상클래스 vs. 인터페이스

- 추상클래스와 인터페이스 모두 new 연산자를 통해 객체화될 수 없음
- 추상클래스
  - ◆ abstract 키워드가 부착된 것과 추상메소드를 포함할 수 있다는 것을 제외하면 일반 실체클래스와 크게 다르지 않음
    - 즉 속성 필드 및 실체메소드 등도 정의 가능
  - ◆ 클래스에서와 마찬가지로 추상클래스도 다중 상속이 허용되지 않음
- 인터페이스 (참조: <https://docs.oracle.com/javase/tutorial/java/landl/interfaceDef.html>)
  - ◆ class 키워드 대신 interface 키워드로 정의됨
  - ◆ 추상메소드, default 메소드, static 메소드 및 상수 필드 포함 가능
  - ◆ 모든 메소드는 public임(생략 가능), 모든 상수필드는 public static final임(생략 가능)
  - ◆ 다중 상속이 허용됨

```
public abstract class Colorable {  
    abstract void setColor(int color);  
}
```

```
public class Car extends Colorable {  
    int color;  
    @Override  
    void setColor(int color) {  
        this.color=color;  
    }  
}
```

```
public interface Colorable {  
    public static final int RED=1;  
    public static final int GREEN=2;  
    public static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public class Car implements Colorable {  
    int color;  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car car=new Car();  
        car.setColor(1);  
    }  
}
```

# 클래스 작성: 인터페이스의 다중 상속

## 인터페이스는 다중 상속 가능

- 다중 상속된 인터페이스 내 동일 명칭 필드는 static이므로 객체와 무관 (예: Colorable.RED, Paintable.RED)

## 클래스는 단일 상속만 허용됨

- 다중 상속이 허용된다면 서로 다른 부모 클래스 내 동일 명칭 필드 접근 시 문제 발생

```
public interface Colorable {  
    static final int RED=1;  
    static final int GREEN=2;  
    static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public interface Movable {  
    void moveTo(int x, int y);  
    void moveBy(int dx, int dy);  
}
```

```
public class Car implements Colorable, Movable {  
    int x, y, color;  
    @Override  
    public void moveTo(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    @Override  
    public void moveBy(int dx, int dy) {  
        this.x+=dx;  
        this.y+=dy;  
    }  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
    @Override  
    public String toString() {  
        return "Color: "+color+", Position="+x+", "+y+"";  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Car car=new Car();  
        car.setColor(Colorable.RED);  
        car.moveTo(100, 100);  
        car.moveBy(50, 70);  
        System.out.println(car);  
    }  
}
```

# 클래스 작성: 추상클래스, 인터페이스의 상속

- 추상클래스나 인터페이스를 상속 받는 자식 클래스를 실제 클래스로 정의할 때
  - 추상클래스나 인터페이스를 상속 받는 자식 클래스에서는 추상 클래스나 인터페이스에 정의된 모든 추상 메소드를 구현해야 한다
    - 구현할 내용이 없다면 메소드 본체를 빈 블록으로 남겨두면 됨 (참조: 아래 예의 moveBy() 참조)
    - 만약 아래 예에서 moveBy()의 구현 내용이 없다고 이 메소드를 삭제한다면 오류 발생

```
public interface Colorable {  
    static final int RED=1;  
    static final int GREEN=2;  
    static final int BLUE=3;  
    void setColor(int color);  
}
```

```
public interface Movable {  
    void moveTo(int x, int y);  
    void moveBy(int dx, int dy);  
}
```

```
public class Car implements Colorable, Movable {  
    int x, y, color;  
    @Override  
    public void moveTo(int x, int y) {  
        this.x=x;  
        this.y=y;  
    }  
    @Override  
    public void moveBy(int dx, int dy) {  
    }  
    @Override  
    public void setColor(int color) {  
        this.color=color;  
    }  
    @Override  
    public String toString() {  
        return "Color: "+color+", Position=("+x+", "+y+")";  
    }  
}
```

# 클래스 작성: 내포 클래스(nested class)

```
public class Employee {
```

```
    class Date {  
        int year, month, day;  
    }
```

```
    String id;  
    Date startDate; // 근무시작일
```

```
    public Employee(String id) {  
        this.id=id;  
        this.startDate=new Date();  
    }  
    public void setStartDate(int year, int month, int day) {  
        this.startDate.year=year;  
        this.startDate.month=month;  
        this.startDate.day=day;  
    }  
}
```

## 내포클래스 (nested class)

- 다른 클래스 내부에 정의된 클래스
- Static nested class vs. non-static nested class

## 내부클래스 (inner class)

- 다른 클래스 내부에 정의된 non-static 클래스

## 로컬클래스 (local class)

- 내부클래스 중 블록(예: 메소드) 내부에 정의된 클래스

## 익명클래스 (anonymous class)

- 내부클래스 중 블록(예: 메소드) 내부에 정의된 이름 없는 클래스

```
public class Test {  
    public static void main(String[] args) {  
        Employee e=new Employee("E-001");  
        e.setStartDate(2019,4,10);  
        System.out.println("근무시작년도: "+e.startDate.year);  
    }  
}
```

# 익명내부클래스를 활용한 코딩

## 익명내부클래스(Anonymous Inner Class)

- 자식 클래스 정의와 그 자식 클래스의 객체 생성을 동시에 작성하는 코드에 사용된 그 자식 클래스

```
public class Animal {  
    String speak(){  
        return "";  
    }  
}
```

```
public class Dog extends Animal {  
    @Override  
    String speak() {  
        return "멍멍";  
    }  
}
```

자식클래스 정의

```
public class Test {  
    public static void main(String[] args) {  
        Dog dog=new Dog(); // 객체 생성  
        System.out.println(dog.speak());  
    }  
}
```

### 실습

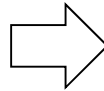
- 아래 코드는 야옹을 출력한다. 익명내부클래스를 이용하여 이 코드를 완성하시오

```
public class Test {  
    public static void main(String[] args) {  
        Animal cat=  
        System.out.println(cat.speak());  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Animal dog=new Animal(){  
            @Override  
            String speak() {  
                return "멍멍";  
            }  
        }; // 자식클래스 정의와 객체생성을 한번에  
        System.out.println(dog.speak());  
    }  
}
```

# 익명내부클래스를 활용한 코딩

```
public interface ActionListener extends EventListener {  
    public void actionPerformed(ActionEvent e);  
}
```



```
public class MyActionListener implements ActionListener {  
    @Override  
    public void actionPerformed(ActionEvent arg0) {  
        JOptionPane.showMessageDialog(null, "부모님, 사랑합니다.");  
    }  
}
```

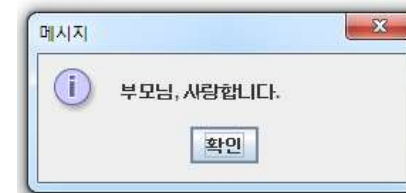
*자식클래스 정의*

```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JButton button=new JButton("Click me");  
        ActionListener actionListener=new MyActionListener();  
        button.addActionListener(actionListener);  
        w.add(button);  
        w.setVisible(true);  
    }  
}
```

*객체생성*

```
public class Test {  
    public static void main(String[] args) {  
        JFrame w=new JFrame();  
        JButton button=new JButton("Click me");  
        ActionListener actionListener=new ActionListener() {  
            @Override  
            public void actionPerformed(ActionEvent e) {  
                JOptionPane.showMessageDialog(null, "부모님, 사랑합니다.");  
            }  
        };  
        button.addActionListener(actionListener);  
        w.add(button);  
        w.setVisible(true);  
    }  
}
```

인터페이스 ActionListener의 자식클래스를 명시적으로 정의하는 대신, 익명내부클래스를 활용하여 자식클래스정의와 객체생성을 한꺼번에 작성하였음





# 인터페이스 활용 예: ActionListener

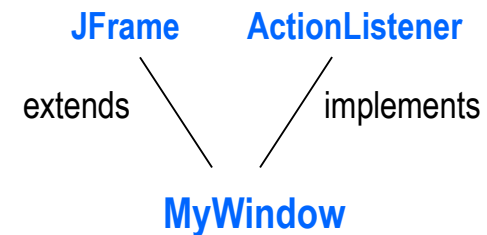
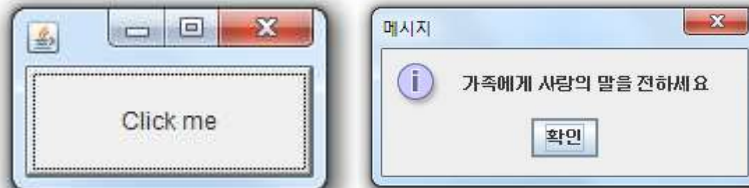
```
public class MyWindow extends JFrame implements ActionListener {
    public MyWindow() {
        JButton button=new JButton("Click me");
        add(button);
        button.addActionListener(this);
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JOptionPane.showMessageDialog(null, "가족에게 사랑의 말을 전하세요");
    }
    public static void main(String args[]){
        MyWindow w=new MyWindow();
        w.setVisible(true);
    }
}
```

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

## MyWindow

JFrame

ActionListener



MyWindow는 JFrame이다  
MyWindow는 ActionListener이다

# 인터페이스 실습 A

- ✚ 인터페이스 ShapeAction을 다음 조건에 따라 정의하시오
  - 실수를 반환하는 추상메소드 getArea()
- ✚ 다음 조건에 따라 클래스 Rectangle(사각형)을 인터페이스 ShapeAction을 구현하여 정의하시오
  - 필드: width(가로 길이, 정수), height(세로 길이, 정수)
  - width, height 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
  - 사각형의 면적을 실수로 반환하는 메소드 getArea()을 ShapeAction의 getArea()를 오버라이드하여 작성
- ✚ 다음 조건에 따라 클래스 Circle(원)을 인터페이스 ShapeAction을 구현하여 정의하시오
  - 필드: radius(반지름, 정수)
  - radius 값을 파라미터로 전달받아 대응하는 필드에 저장하는 생성자
  - 원의 면적을 실수로 반환하는 메소드 getArea()을 ShapeAction의 getArea()를 오버라이드하여 작성
- ✚ 클래스 Test의 main() 내에서 다음 절차를 코딩하시오
  - 다음 객체들을 **ShapeAction** 배열에 저장
    - ◆ Rectangle 객체(가로 3, 세로 4), Circle 객체(반지름 5), Circle 객체(반지름 2)
  - 반복문을 통해 배열에 저장된 각 도형의 면적(getArea() 호출)을 출력

# 클래스, 객체

익셉션, try-catch, checked vs unchecked 익셉션, Wrapper 클래스, 박싱, 언박싱, 제네릭 클래스, LinkedList, HashMap

# 익셉션(Exception)

## ✚ 익셉션(예외, exception)

- 프로그램 실행 도중에 정상적인 실행을 제한하는 예외 상황을 지칭하는 용어

## ✚ 익셉션 발생 시 적절한 처리가 요구됨

```
public class Test {  
    public static void main(String[] args) {  
        int      n1=0, n2=5;  
        System.out.println(n2/n1);  
    }  
}
```

divide by zero

<실행결과>

Exception in thread "main"

java.lang.ArithmeticException: / by zero

```
public class Test {  
    public static void main(String[] args) {  
        User user=new User("gdhong", "홍길동", "1234");  
    }  
}
```

← 비밀번호

- User 객체 생성 규정 → User 객체의 비밀번호 길이는 **최소 6글자 이상** 되어야 함
- "1234"는 위의 비밀번호 규정을 만족하지 못하므로 User 객체는 생성되면 안 되므로 이러한 예외 상황에 대한 처리가 필요함

# 익셉션 처리: try-catch 절 사용

## 익셉션 처리 미적용

```
public class User {  
    String id;  
    String name;  
    private String password;  
    public User(String id, String name, String password) {  
        this.id=id;  
        this.name=name;  
        this.password=password;  
    }  
    @Override  
    public String toString(){  
        return id+","+name;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        User user=new User("gdhong", "홍길동", "1234");  
        System.out.println(user);  
    }  
}
```

## 익셉션 처리 적용

```
public class User {  
    String id;  
    String name;  
    private String password;  
    public User(String id, String name, String password) throws Exception {  
        if(password.length()<6) throw new Exception("패스워드는 6 문자 이상");  
        this.id=id;  
        this.name=name;  
        this.password=password;  
    }  
    @Override  
    public String toString() {  
        return id+","+name;  
    }  
}  
  
public class Test {  
    public static void main(String[] args) {  
        try {  
            User user= new User("gdhong", "홍길동", "1234");  
            System.out.println(user);  
        } catch (Exception e) {  
            System.out.println(e.getMessage());  
        }  
    }  
}
```

Exception이 발생할 수 있는 메소드라는 표시

예외 상황 감지 및 익셉션 발생 시킴

Exception이 발생할 수 있는 메소드를 호출하는 경우 반드시 **try {} catch {}** 절로 감싸야 한다

# Checked vs. Unchecked Exception

## ✚ Checked 익셉션

- 컴파일 시점에 check(검사)되는 익셉션
- Checked 익셉션을 발생시키는 메소드를 호출하는 경우
  - ◆ try-catch 절이나 throws 절을 사용하여 checked 익셉션 처리해야 함

## ✚ Unchecked 익셉션

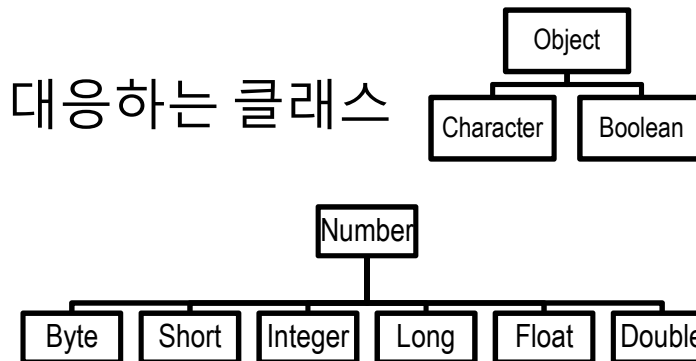
- 컴파일 시점에 check(검사)되지 않는 익셉션
  - ◆ 예) ArithmeticException, ArrayIndexOutOfBoundsException, NullPointerException, NumberFormatException

# Wrapper 클래스

## Wrapper 클래스

- 기본 자료형(primitive data type)에 대응하는 클래스

기본 자료형	Wrapper 클래스
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean



```
public class Test {  
    public static void main(String[] args) {  
        Integer n=new Integer(95);  
        String v="점 수="+n.toString();  
        System.out.println(v);  
  
        int score=3+n.intValue();  
        System.out.println(score);  
    }  
}
```

# Autoboxing, unboxing (박싱, 언박싱)

## Boxing (박싱)

- 기본 자료형 값을 대응하는 Wrapper 클래스 객체로 변환하는 작업

## Unboxing (언박싱)

- Wrapper 클래스 객체를 기본 자료형 값으로 변환하는 작업

```
public class Test {  
    public static void main(String[] args) {  
        // autoboxing  
        Integer n=95; // Integer n=new Integer(100);  
  
        // unboxing  
        int m=n; // int m=n.intValue();  
  
        System.out.println(n); // println(Object x)  
        System.out.println(m); // println(int x)  
    }  
}
```

## 실습

- Autoboxing을 이용하여 3.14를 Double형 참조 변수 pi에 저장하는 코드를 작성하시오
- 변수 pi에 저장된 값을 double 변수 x에 저장하는 코드를 작성하시오

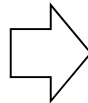
## 실습

- Autoboxing을 이용하여 1~10까지의 정수를 Integer 배열 n에 저장하는 코드를 작성하시오



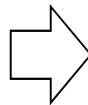
# Generic class (제네릭 클래스)

```
public class Player {  
    int record;  
    public void setRecord(int record) {  
        this.record = record;  
    }  
    public int getRecord() {  
        return record;  
    }  
}
```



```
public class Player<T> {  
    T record;  
    public void setRecord(T record) {  
        this.record = record;  
    }  
    public T getRecord() {  
        return record;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
  
        Player p1=new Player();  
        p1.setRecord(32); // 32점  
        System.out.println(p1.getRecord());  
  
        Player p2=new Player();  
        p2.setRecord(9.875); // 9.875초  
    }  
}
```



```
public class Test {  
    public static void main(String[] args) {  
        Player<Integer> p1=new Player<Integer>();  
        p1.setRecord(32); // 단일 경기 최다 점수 기록 32점  
        System.out.println(p1.getRecord());  
        Player<Double> p2=new Player<Double>();  
        p2.setRecord(9.875); // 100미터 기록 9.875초  
        System.out.println(p2.getRecord());  
        Player<Integer []> p3=new Player<Integer []>();  
        Integer v[]={210, 195, 220}; // 높이뛰기 기록  
        p3.setRecord(v);  
        Integer w[]=p3.getRecord();  
        for (int i = 0; i < w.length; i++) System.out.println(w[i]);  
    }  
}
```

현재 Player 클래스를 통해서는 정수, 실수, 정수 배열 등 다양한 유형의 기록 자료들을 저장하기 어려움

오류

# JDK 제네릭 클래스 사용 예: LinkedList

```
public class Test {  
    public static void main(String[] args) {  
        LinkedList<String> country=new LinkedList<>();  
        country.add("한국");  
        country.add("미국");  
        country.add("일본");  
        System.out.println(country);  
    }  
}
```

## 실습

- 다음 점수 값들의 리스트 [85, 91, 73]를 LinkedList 객체로 생성 후 출력하시오.
- 요일 문자들의 리스트 [월,화,수,목,금,토,일]를 LinkedList 객체로 생성 후 출력하시오.
- 다음 기록 값들의 리스트 [12.9, 9.98, 10.52]를 LinkedList 객체로 생성 후 출력하시오.

```
public class Student {  
    String id;  
    int score;  
    public Student(String id, int score) {  
        this.id=id;  
        this.score=score;  
    }  
    @Override  
    public String toString() {  
        return "<" + id + ", " + score + ">";  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s=new Student("s-001", 98);  
        LinkedList<Student> list=new LinkedList<>();  
        list.add(s);  
        list.add(new Student("s-002", 100));  
        System.out.println(list);  
    }  
}
```

# JDK 제네릭 클래스 사용 예: HashMap

```
public class Test {  
    public static void main(String[] args) {  
        HashMap<String, Integer> rank=new HashMap<>();  
        rank.put("한국", 7);  
        rank.put("중국", 10);  
        rank.put("일본", 25);  
        System.out.println(rank);  
    }  
}
```

## 실습

- <국가, 수도>의 쌍 <한국, 서울>, <중국, 베이징>, <일본, 도쿄>들을 HashMap 객체로 생성 후 출력하시오.  
(국가명, 수도명을 각각 key, value로 설정)


```
public class Student {  
    String id;  
    int score;  
    public Student(String id, int score) {  
        this.id=id;  
        this.score=score;  
    }  
    public String getId() {  
        return id;  
    }  
    @Override  
    public String toString() {  
        return "<"+id+", "+score+">";  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Student s1=new Student("s-001", 98);  
        Student s2=new Student("s-002", 100);  
        HashMap<String, Student> map=new HashMap<>();  
        map.put(s1.getId(), s1);  
        map.put(s2.getId(), s2);  
        System.out.println(map);  
    }  
}
```

## References

 <http://docs.oracle.com/javase/7/docs/api/>

 <https://docs.oracle.com/javase/tutorial/java/>

 김윤명. (2008). 뇌를 자극하는 Java 프로그래밍. 한빛미디어.

 남궁성. 자바의 정석. 도우출판.

 황기태, 김효수 (2015). 명품 Java Programming. 생능출판사.