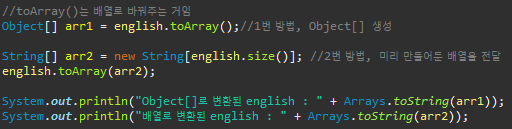
#Random 클래스  

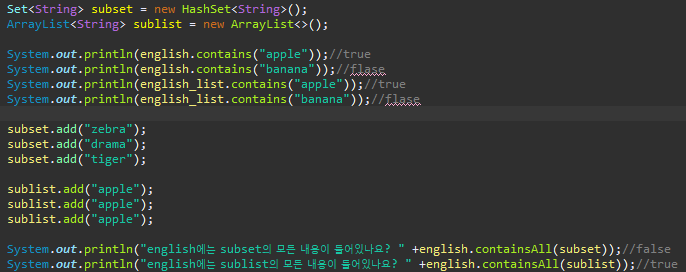

#Arrays.toString(배열의 이름) : 배열 편하게 출력하기

#String.format : 서식을 통해 내가 원하는 문자열을 만드는 것  
ex. String.*format*("서식 %d", 10);

**\*컬렉션들을 배열로 변환하기**

\*contains(): 컬렉션에 해당 요소가 포함되어 있는지 여부를 반환

\*containAll(Collection) : 전달한 컬렉션이 모두 포함되어 있는지를 검사



\*addAll(Collection) : 전달한 컬렉션의 내용을 모두 추가(추가 성공 여부를 반환🡪리턴 타입 boolean)**🡪단 제네릭 타입이 같아야함**  
-추가 성공 여부를 반환  
-이미 모두 포함된 경우에는 false를 반환

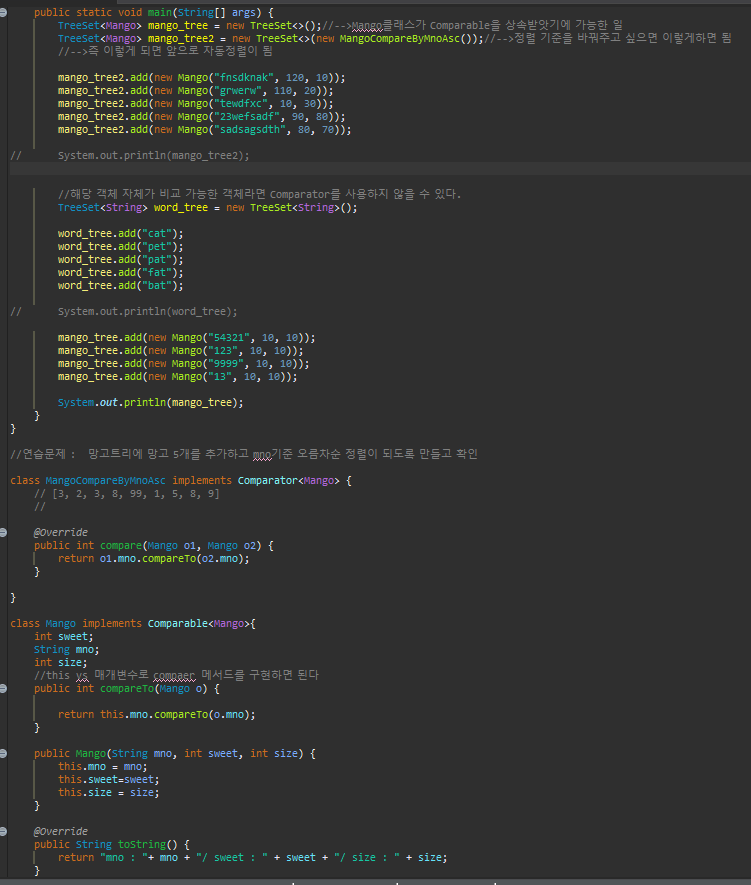
\*removeAll(Collection) : 전달한 컬렉션과 같은 내용을 모두 제거  
-지워진게 있으면 true 없으면 false를 반환

\*retainAll(Colleciont) : 전달한 컬렉션과 일치한 내용만 남긴다.  
-변한 것이 있으면 true, 없으면 false를 반환  
**\*isEmpty() : 해당 컬렉션이 비어있으면 true, 아니라면 false를 반환**

#문자열을 숫자로 변환  
Integer.*parseInt,* ("1234", 10(진법);  
Character.*toString*(str.charAt(i) 🡪문자열을 문자(String)으로 변환해줌

# compare 메서드를 통해 Grape 클래스의 크기를 결정 하고 Collections.sort(grapes(리스트 이름), Comparator 객체)를 해주면 정렬을 해줌

\*Comparator : 비교할때 사용하는 비교 도구 클래스로 만드는 인터페이스

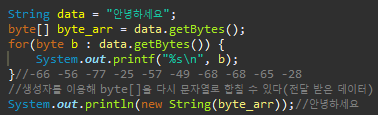
\*Comparable : 이 객체가 비교 가능한 클래스임을 표시하는 인터페이스  
  
#반복문으로 Map을 활용하기

* keySet() : key들로만 이루어진 Set을 반환한다
* values() : value들로만 이루어진 Set을 반환한다.
* **entrySet() : Entry<key, value>로 이루어진 Set을 반환한다  
  (entry란 하나의 key와 value의 쌍이라고 생각하면 됨)**
* containsKey(key) : 맵에 해당 키가 포함되어 있는지 검사한다(boolean 타입)
* containsValue(value) : 맵에 해당 값이 포함되어 있는지 검사한다(boolean 타입)

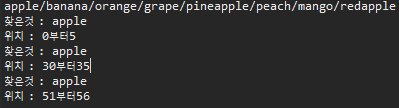
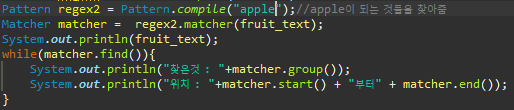
# **A instanceof B : A가 B의 인스턴스인지를 판별한다. 업캐스팅된 상태에서도 판별된다.**



**#String**

* split(delimiter) : 문자열을 원하는 기준으로 자른 후 문자열 배열로 변환  
  
* join(delim, 이어붙이고싶은 것(배열, 컬렉션이 들어갈 수 있음)) : 뒤에 나오는 문자열들을 모두 delim으로 이어붙인다.  
  ****
* Substring(start(int), end(int)) : start부터 end미만까지 문자열을 잘라서 반환
* Replace(“바꿔야할 문자열”, “바꾸고싶은 문자열”) : 문자열을 치환한다.(**정규표현식도 가능**)  
  ****
* Format(format, value) : 서식을 이용해 원하는 문자열을 생성  
  
* Contains(seq) : 문자열에 해당 문자열이 포함되어 있는지 검사한다  
  ****
* Stateswith(seq) : 문자열이 해당 문자열로 시작하는지 검사한다(경로 검사에 유용)
* endsWith(seq) : 문자열이 해당 문자열로 끝나는지 검사한다(확장자 검사에 유용)
* 
* getBytes(): 문자열을 byte[]로 변환한다 (데이터 전송 준비)  
  이때 byte[]로 나뉠 때와 같은 인코딩 타입을 사용해야 한다.  
  
* trim() : 문자열 바깥쪽의 공백을 제거해 준다.  
  ****
* toUpperCase() : 문자열을 모두 대문자로 변환한 새로운 인스턴스 반환
* toLowerCase() : 문자열을모두 소문자로 변환한 새로운 인스턴스 반환
* equalsIgnoreCase() : 대소문자 상관없는 equals()
* getChars(start, end, 담을곳, 담을곳 시작지점) : char[]에 문자열의 원하는 부분을 담을 수 있다

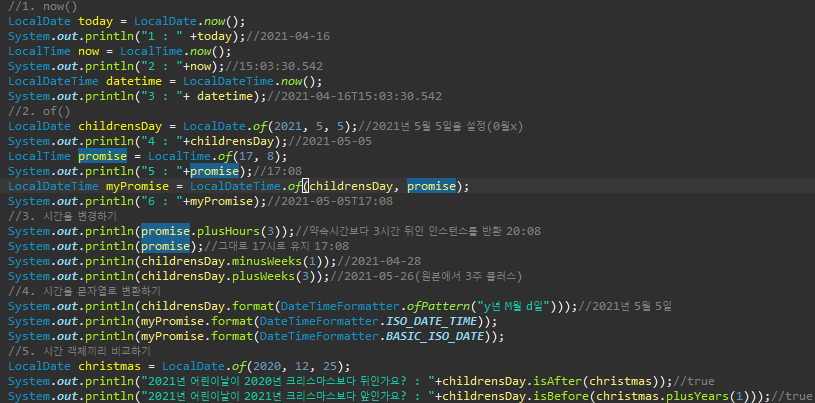
**#split() : 컴파일된 패턴 인스턴스로 문자열 split()을 진행한다.**🡪**/을 기준으로 잘라줌**

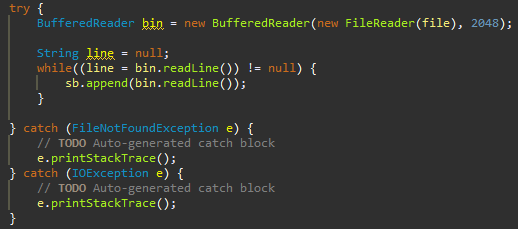
**#matcher() : Matcher 인스턴스를 생성하여 반환한다**  
🡪즉 기준을 찾아줌

**#Time**  
\*LocalDate: 날짜만 저장하기 위한 클래스

LocalTime : 시간만 저정하기 위한 클래스

LocalDateTime : 날짜와 시간을 모두 저장하기 위한 클래스



\*BufferedReader로 파일을 읽어오는 방법 (\*얘가 배열로 받는 것 보단 더 좋음), 1줄씩 읽기  


\*BufferedWriter 로 파일을 쓰는 방법  

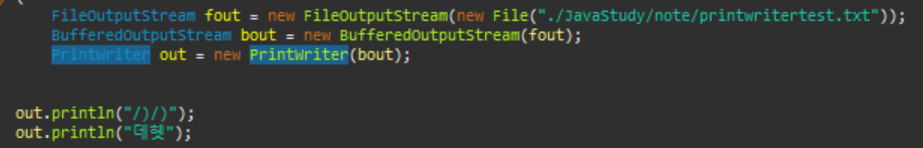

21.05.18

**-Character.isAlphabetic(char) : char이 알파벳인지 검사해주는 메서드(bollen 리턴)**

**-Character.getType(char) : char이 어떤 타입인지 (int 리턴-대문자 : 1, 소문자 : 2)   
(**Character.***UPPERCASE\_LETTER*: 1,** Character.***LOWERCASE\_LETTER*: 2)**

  
🡪위 내용을 활용한 거임

**#PrintWriter**

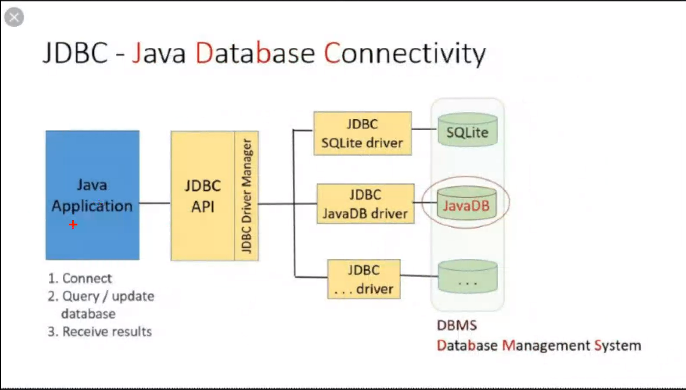
* Writer를 한 번 더 감싸서 편리한 기능(print 메서드)까지 추가해놓은 클래스
* Printf(), println()등이 구현되어 있다  
  

**#PrintStream**

* Stream을 한 번 더 감싸서 편리하게 print를 할 수 있는 클래스  
  (Stream을 감싸지 않으면 123을 넣으면 문자로 인식하여 넣을 수도 있음)

21.05.31

**# JDBC**

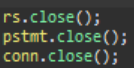
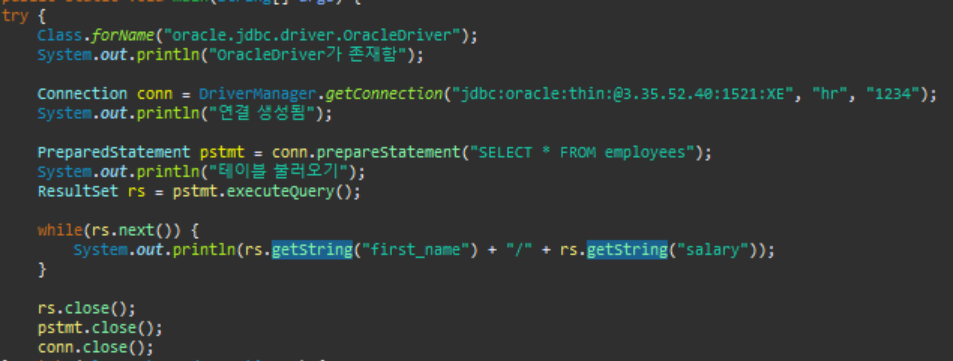
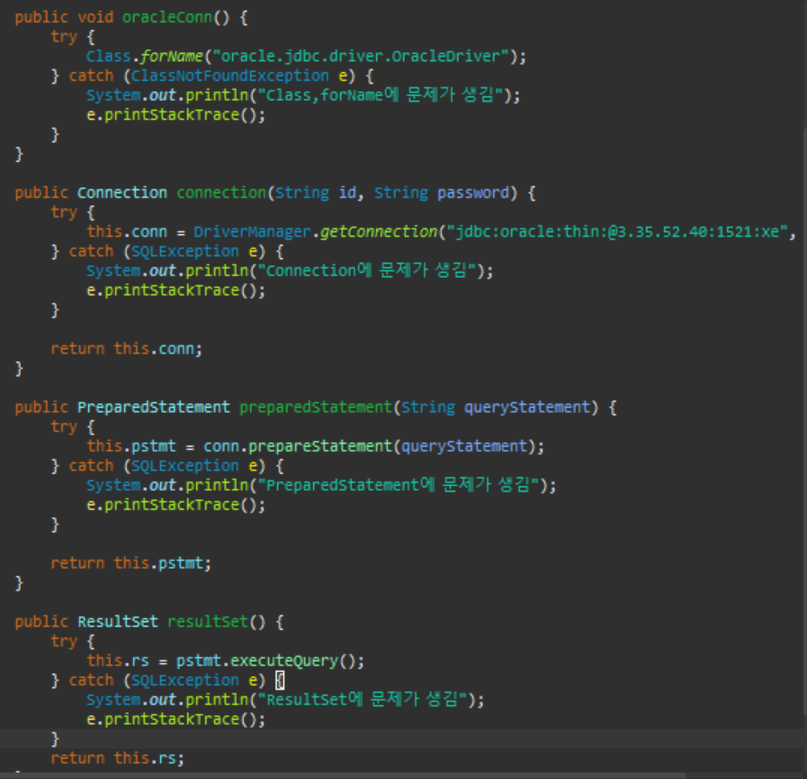
* Java Database Connectivity
* DB에 쿼리문을 전달하여 실행하고 결과를 받아오는 Java API
* 어떤 DB와도 호환할 수 있도록 설계되어 있다
* Java 부분의 소스는 어떤 DB를 사용하더라도 항상 일정하다
* DB를 제작한 쪽에서 jdbc에서 요구하는 형식을 맞추어 개발하기 때문에 프로그래머들은 새로운 DB도 기존의 jdbc로 편리하게 사용할 수 있다.  
  

**# OJDBC**

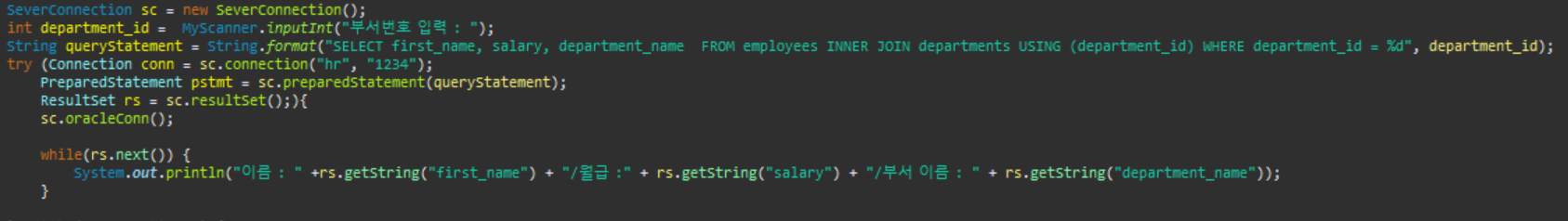
* 오라클 데이터베이스가 JDBC API와 소통하기 위해 만든 라이브러리
* 오라클 DB가 설치된 폴더에 포함되어 있다

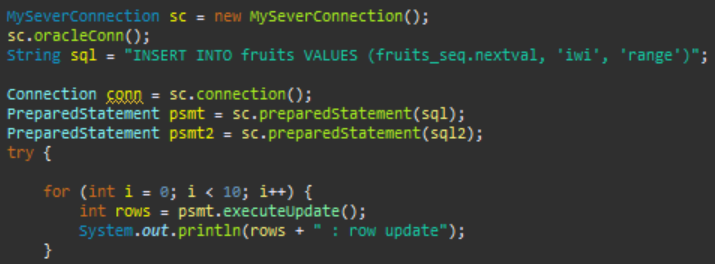
프로젝트 우클릭-Build Path- Configure Bulid Path - Libraries -Add External JARs - sqldeveloper/jdbc/lib에 잇는 jar를 추가  
🡪라는 폴더가 해당 프로젝트 안에 생성됨  
🡪이게 있어야 오라클과 연동 가능(없어도 되긴 하나 경로가 복잡함)

\*어떤 회사 DB를 사용하더라도 거쳐야 하는 똑 같은 절차

1. Class.forName(driverName)으로 해당 DB사의 jdbc드라이버를 로드한다(driverName에 class이름을 써서 클래스도 불러올 수 있음)  
   Class.*forName*("oracle.jdbc.driver.OracleDriver");  
   **🡪생략가능!!**
2. DriverManager 클래스를 통해 DB와의 연결을 생성한다  
   DriverManager.*getConnection*(url, user, password);  
   Connection conn = DriverManager.*getConnection*("jdbc:oracle:thin:@3.35.52.40:1521:XE", "hr", "1234");
3. 생성된 연결을 통해 전송할 쿼리문을 생성한다  
   PreparedStatement pstmt = conn.prepareStatement("SELECT \* FROM employees");
4. 생성된 쿼리문을 DB로 보낸다  
   ResultSet rs = pstmt.executeQuery();
5. 받아온 결과를 마음대로 사용한다   
     
   **🡪즉 rs.getString(“컬럼이름”);을 통해 순차적으로 받아온다**
6. 다 썼으면 연결을 잘 끊어준다(가장 나중에 열었던 순서대로 닫는다  
     
   예시)  
    🡪서버 관련 메서드들을 만든거임

\*JDBC에서 SELECT는 ResultSet을 반환한다.그 외의 DML(INSERT, UPDATE, DELETE)는 적용된 행의 개수를 반환한다

🡪이를 통해   
**이와 같이 만들어진 메서드를 사용하며 try close를 이용하여 자동으로 close하게 한다!  
또한 컬럼의 순서를 안다면**System.***out***.println("이름 : " +rs.getString(1) + "/월급 :" + rs.getString(2) + "/부서 이름 : " + rs.getString(3));  
**처럼 컬럼 이름을 넣는 것이 아닌 컬럼 순서 넘버로 출력이 가능하다**

\*SELECT는 PrepparedStatement의 executeQuery()메서드를 사용하고 나머지는 executeUpdate()메서드를 사용한다  


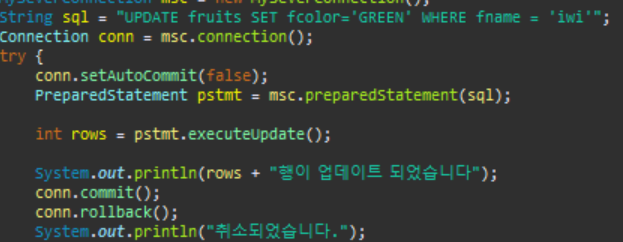
🡪입력은 executeUpdate로 하며 1행이 입력될 때마다 1이 반환됨

또한 시퀀스를 통해 FID를 1씩 증가시키며 입력(이때 나머지 데이터는 동일하게 들어감…)

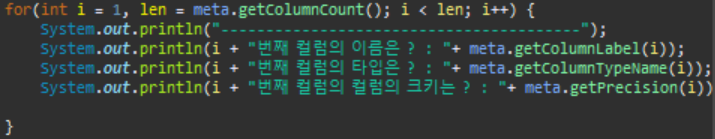
\*JDBC로 처음 열결을 생성하면 기본적으로 AutoCommit이 활성화 되어있다.

\*트랜젝션을 직접 제어하기 위해서는 자동 커밋을 비활성화 해야 한다.  
conn.setAutoCommit(false);  
**🡪따라서 트랜잭션을 직접 제어하기 위해서는 위와 같은 코드는 필수**

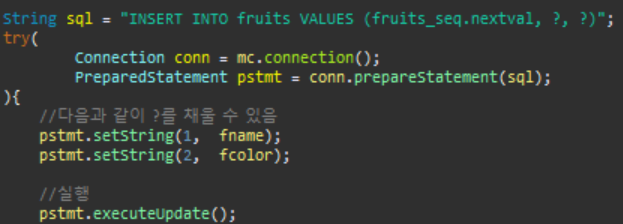
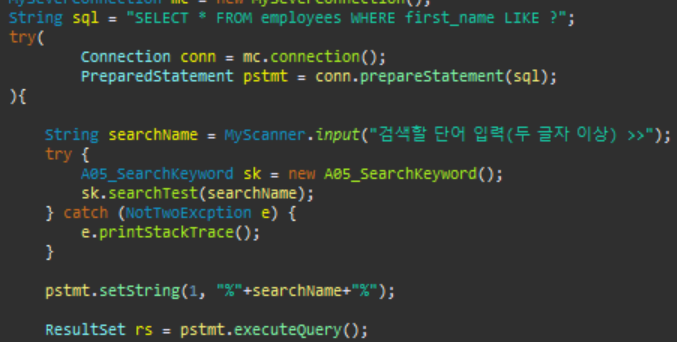
conn.commit();🡪커밋해주는 코드

conn.rollback();🡪롤백해주는 코드  


21.06.01

\*가져온 데이터의 메타 데이터들을 보고 싶을 때  
ResultSetMetaData meta = rs.getMetaData();  
System.***out***.println("컬럼이 몇 개인가? : "+ meta.getColumnCount());  
  
\*getColumnLabel : 컬럼의 이름, getColumnTypeName : 컬럼의 타입, getPrecision : 컬럼의 컬럼 크기(NUMBER(5)이면 크기는 5)

\*   
🡪이렇게 쓰면 보완성에 취약함

**\*PrepareStatement에 준비시킬 쿼리문에 변수가 필요하다면 변수자리는 ?로 채운다  
  
🡪단 setString으로 넣을땐 자동으로 ‘’처리를 해줌!  
**

**#ConnectionPool**

* DB에 DML을 적용할때마다 새로운 접속을 생성하는 것은 리소스가 많이 낭비된다.
* 연결을 이미 여러 개 생성해두고, 사용중이지 않은 커넥션을 꺼내주는 방식을 커넥션 풀이라고 한다.

**\*구글-hikariCP-밑으로 내려가면 OR download from here 들어가서-최선꺼 jar 다운-넣고 싶은데 넣기**  
(난 C:\Users\xoghk\OneDrive\바탕 화면\공부\국비\git\FirtsRepo\JavaStudy\files다가 넣음)

그리고

HikariConfig config = new HikariConfig();

config.setJdbcUrl("url");

config.setUsername("id");

config.setPassword("pw");

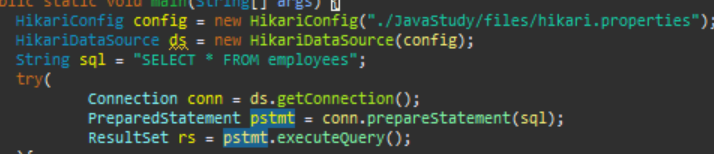
config.addDataSourceProperty("cachePrepStmts", "true");

config.addDataSourceProperty("prepStmtCacheSize", "250");

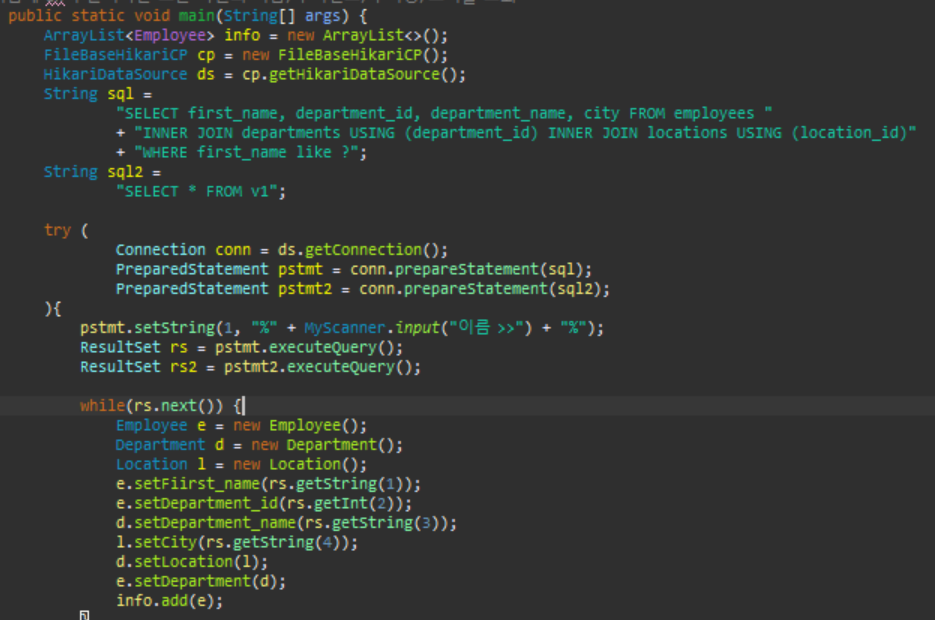
config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");

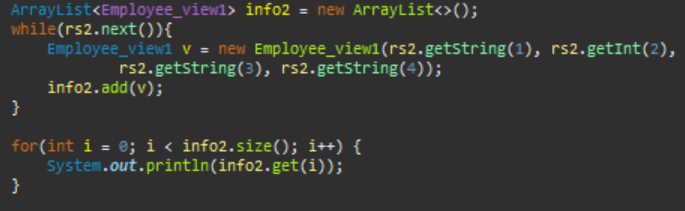
를 복사해서 붙여아함

HikariDataSource ds = new HikariDataSource(config);

그리고 org.slf4j.LoggerFactory 라이 브러리를 다운 받아야함  
<https://mvnrepository.com/artifact/org.slf4j/slf4j-api/1.7.30>  
🡪여기 있음  
--slf4j : 로그를 찍는 용도로 사용하는 인기 라이브러리  
  
🡪이렇게 쓰는 거임

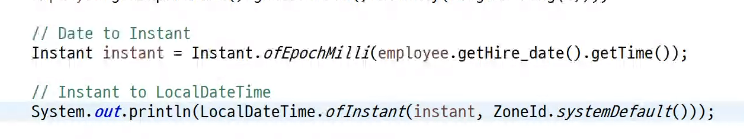
21.06.02

\*뷰를 통해 원하는 데이터들로만 만들어서 모델 클래스에 넣기  
  
🡪조인을 하여 Employee, Department, Location 모델 클래스에 넣는 방법(Employee안에 Department 클래스가 존재하고 Department 안에 Location 클래스 존재)  
🡪혹은 pstmt2는 뷰를 만들어 원하는 정보만 얻는 방법

\*tip – 원래 모델 클래스를 상속받아서 뷰에 맞는 생성자만 만들어 내는 방법도 있음  
  
🡪원래 모델 클래스를 상속받은 자식 클래스  
🡪

🡪자식 클래스에 뷰의 내용을 집어넣어서 toString으로 출력  
🡪  
🡪다시 공부하기(구글에 date to localdatetime 검색 ㄱㄱ)

Instant instant = Instankt.ofEpochMilli(employee.getHire\_date().getTime()));

LocalDataTime.of(Instant(instant, ZoneId.systemDefault()));  


**#Swing**

* Java로 윈도우 GUI 프로그램으ㄹ 만들 수 있는 API

**#Container**

* 컴포넌트를 추가할 수 있는 판
* Jframe, Jpanel, Jdialog …등
* 컨테이너마다 원하는 레이아웃을 지정할 수 있다.

**#Component**

* 컨테이너에 추가할 수 있는 부품
* Jbutton, Jtable, Jslider, Jmenu …등

**#Jframe**

* 프레임 컨테이너. 다양한 컴포넌트들을 추가할 수 있다.

**Jframe을 상속 받고 하자**  
public class S02\_JButton extends JFrame{

\*X버튼을 눌렀을 때의 동작 설정  
setDefaultCloseOperation(JFrame.***EXIT\_ON\_CLOSE***);

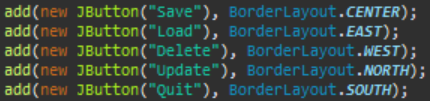
\*프레임을 보이게 설정  
setVisible(true);

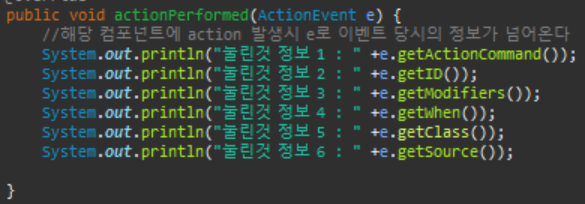
\*프레임 크기 설정  
setSize(600, 600);

\*위치 설정  
setLocation(200, 50);

\*프레임의 레이아웃 설정  
setLayout(new BorderLayout());  
(border말고 다른 것도 있음)

**#BorderLayouy**

* 동서남북과 가운데로 이루어진 레이아웃
* 컴포넌트를 추가할 때 위치를 함께 지정해 줄 수 있다
* 수평 여백과 수직 여백을 설정할 수 있다.  
    
  quit\_btn.setFont(new Font("궁서체", Font.***BOLD***, 35));  
  🡪이런식으로 각 컴포넌트를 추가할 수 있으며 컴포넌트에 대한 설정도 하면 됨

\*버튼에 이벤트를 추가  
quit\_btn.addActionListener(new ActionListener()  


\*e.getSource()에는 액션이 발생한 컴포넌트의 인스턴스가 들어있고 다운캐스팅 하여 사용해야 한다  
((JButton) e.getSource()).setText("눌렸음");