1. 예외(에러, 오류)

1. 프로그램이 동작하다 어떠한 원인에 의해서 프로그램 강제 종료되는 경우가 있는데 이런 경우를 초래하는 원인을 에러, 오류, 예외라고 부른다.
2. 컴파일에러 : 컴파일 중 발생한 에러
3. 런타임에러 : 프로그램 실행 중 발생한 에러
4. 논리적에러 : 실행은 되는 데 생각대로 동작하지 않는 것
5. 위처럼 발생한 에러들에 의해 프로그램이 강제 종료되는 것을 막거나 에러를 로그로 남겨서 에러가 발생하지 않도록 소스코드를 수정하는 행위
6. Checked Exception : 컴파일러가 예외 처리를 강제시키는 예외(무조건 예외처리 코드를 작성해야되는 예외). IOException, ClassNotFoundException .. throws, try~catch 블록으로 예외처리.
7. Unchecked Exception : 컴파일러가 예외 처리를 강제하지 않는 예외(예외처리 코드를 작성하지 않아도 된다). NullPointerException, ArrayIndexOutOfBoundException...

3. try~catch~finally 블록

1. try{} 블록안의 소스코드를 실행하다가 예외가 발생하면 catch{} 블록으로 이동해서 예외처리코드를 실행하는 코드.

```
try{
   try구문의 소스코드 실행 중에 예외가 발생하면 catch블록으로
} catch(Exception e) {
   예외처리 코드작성
} finally {
   예외가 발생하던 발생하지 않던 무조건 실행
}
```

```
try{
   try구문의 소스코드 실행 중에 예외가 발생하면 catch블록으로
} catch(ClassNotFoundException ce) {
   ClassNotFoundException 대한 예외처리 코드작성
} catch(NullPointerException ne) {
   NullPointerException 대한 예외처리 코드작성
} catch(Exception e) {
   위에 ClassNotFoundException, NullPointerException 두 예
} finally {
   예외가 발생하던 발생하지 않던 무조건 실행
}
```

2. finally{} 블록은 예외가 발생하던 발생하지 않던 무조건 실행되는 블록.
3. Exception의 getMessage(), printStackTrace()
    - getMessage() : 발생한 예외 클래스의 객체에 저장된 메세지를 전달.
    - printStackTrace() : 예외발생 시 호출된 메소드 정보와 클래스 정보 그리고 예외메세지까지 모두 출력. 보안상의 문제로 취약점에 무조건 걸려서 println()이나 log 라이브러리로 대체해서 사용한다.
4. 예외가 발생했을 때 try{} 블록의 코드 실행 -> 코드 실행중 예외발생시 catch{} 블록으로 이동 -> catch{} 블록에서 예외처리 -> finally{} 블록 실행
5. 예외가 발생하지 않을 때 try{} 블록의 코드 실행 -> finally{} 블록 실행
6. finally{} 블록의 작성여부는 선택사항이지만 try{}~catch{}는 무조건 작성

4. 예외 발생시키기

1. 예외클래스 변수명 = new 예외클래스(메세지); throw 변수명;

5.메소드에서 예외 선언(throws)

1. 메소드를 호출한 곳에 예외를 전달.
2. 리턴타입 메소드명(매개변수) throws 예외클래스 {

}

6. 예외처리의 중요성

1. 예외처리는 프로그램의 안정성 보장해줄 수 있는 코드이므로 습관적으로 작성할 수 있도록 연습한다. 예외처리 코드를 작성하여 예외발생하여 프로그램이 안정적 돌아갈 수 있도록 한다.
2. try{}~catch{}~finally{} 분리되어 있기 때문에 코드 가독성을 높인다. 블록별로 나눠져있기 때문에 코드의 흐름을 파악하기 쉽다.