

# 예외처리 (Exception Handling)

## 오류의 종류

*NullPointerException*

- 컴파일 에러(Compile-Time Error) : 소스 상의 문법 Error.
- 런타임 에러(Runtime Error) : 입력 값이 틀렸거나, 배열의 인덱스 범위를 벗어났거나, 계산식의 오류 등으로 인해 발생함
- 논리 에러(Logical Error) : 문법상 문제가 없고, 런타임에러도 발생하지 않지만, 개발자의 의도대로 작동하지 않음.
- 시스템에러(System Error) : 컴퓨터 오작동으로 인한 에러 -> 소스 구문으로 해결 불가

# 자바 오류 정의

## Error(오류)

new

프로그램 수행시 치명적 상황이 발생한 것으로 소스상 해결이 불가능 한 것을 에러(Error)라고 한다. 프로그램이 비정상적인 종료됨.

OutOfMemoryError, StackOverflowError 등.

## Exception(예외)

프로그램 오류중 적절한 코드에 의해서 수습될 수 있는 미약한 오류.

예외발생상황을 예측해서 미리 예외처리코드를 작성해둔다.

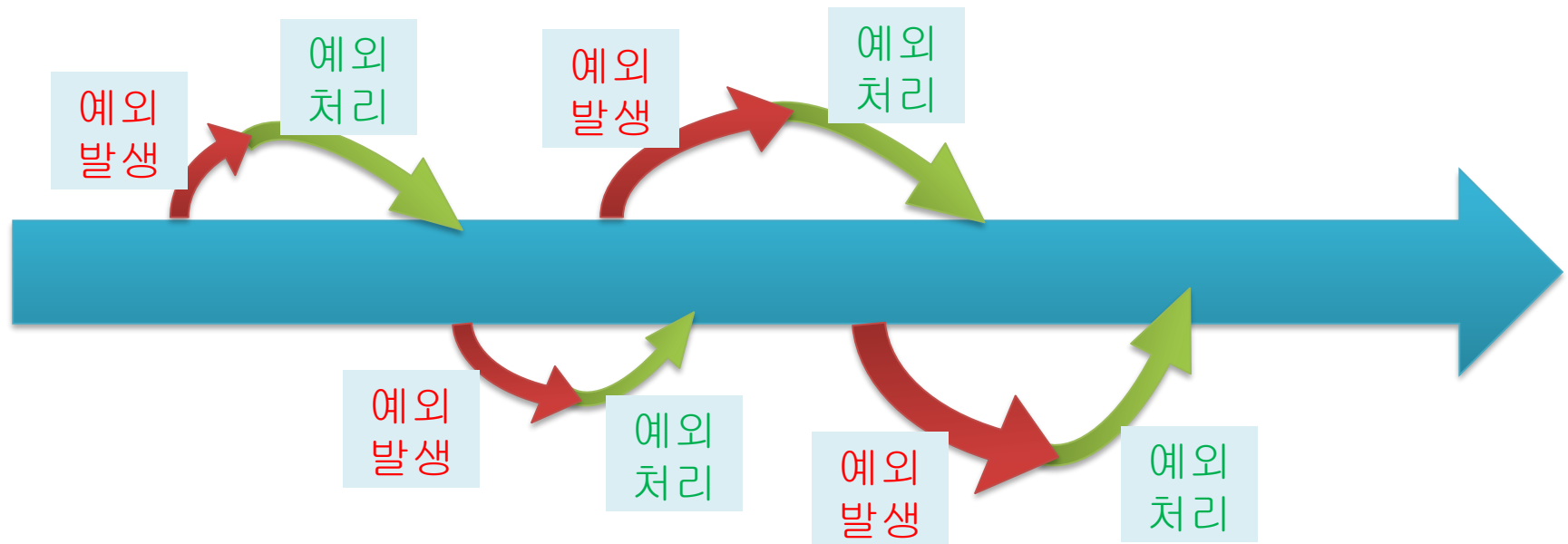
NullPointerException, ArithmeticException, IOException 등.

# 예외처리(try~catch)의 목적

## 이걸 대체 왜 하느냐?

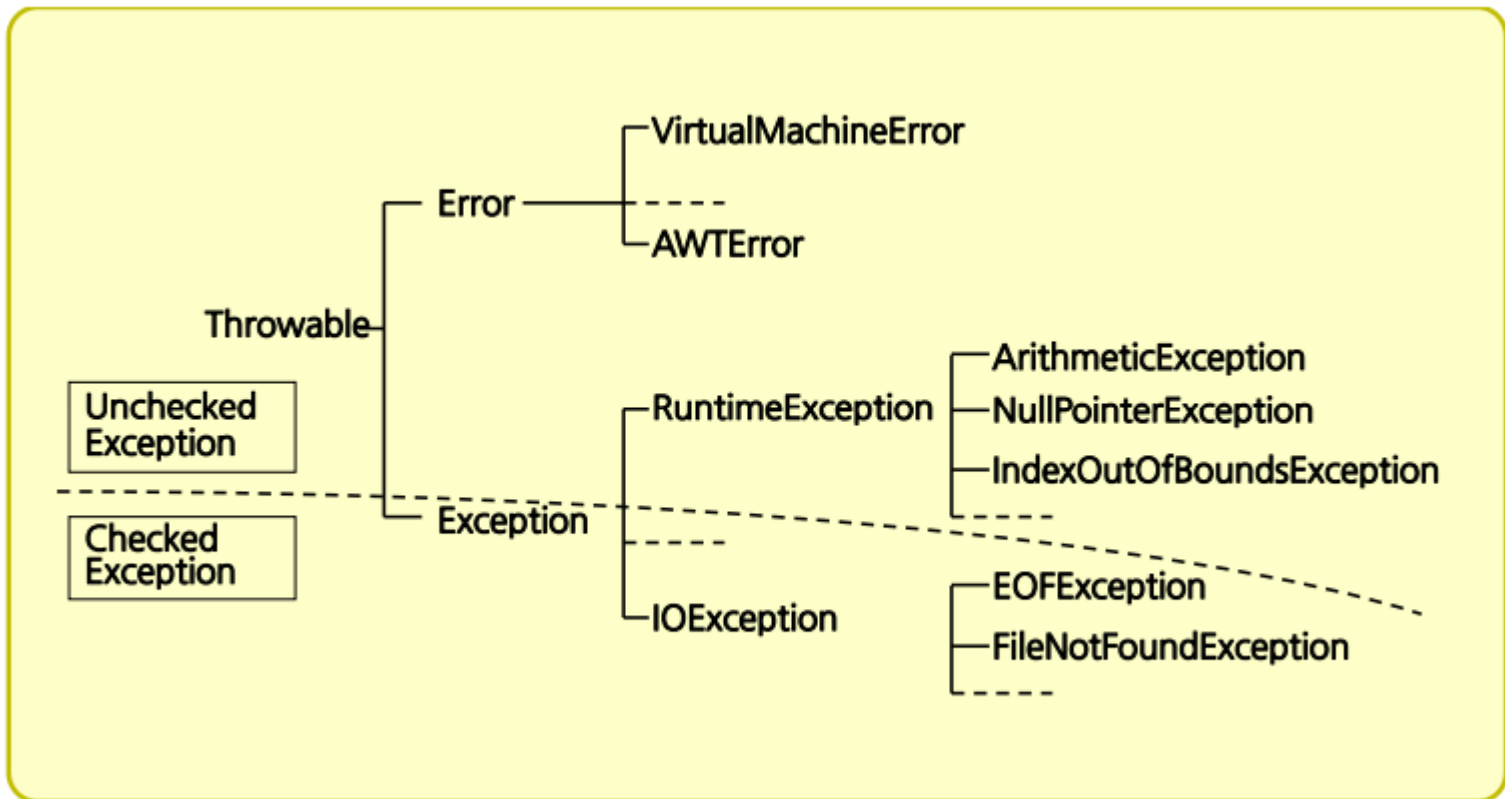
목적 : 프로그램의 비정상 종료를 막고, 정상적인 실행상태를 유지함.

어떻게: 예외상황이 발생된 경우에 처리로직을 만듦.



## Exception 클래스 상속도

소스코드상에서 반드시 개발자가 처리해야 되는 **Checked Ecxeption**과 개발자가 소스 코드 상에서 다룰 필요가 없는 **Unchecked Exception** 두 부류로 나누어진다.



# RuntimeException 클래스

Unchecked Exception이며, 주로 프로그래머의 부주의로 인한 bug인 경우가 많기 때문에 Exception처리보다는 코드를 수정해야 하는 경우가 많다.

예외처리를 강제화 하지 않는다.

## ▪ ArithmeticException

- 0으로 나누는 경우에 발생
- If문으로 먼저 나누는 수가 0인지를 검사해야 함

## ▪ NullPointerException

- Null인 ref.변수로 객체 멤버 참조 시도
- 객체를 사용하기 전에 ref.변수가 null인지 먼저 확인해야 함

## ▪ NegativeArraySizeException

- 배열 크기를 음수로 지정한 경우
- 배열 size를 0보다 크게 지정해야 함

## ▪ ArrayIndexOutOfBoundsException

- 배열의 index범위를 넘어서 참조하는 경우
- 배열이름.length를 써서 배열의 범위를 확인해야 함

## ▪ ClassCastException

- Cast연산자 사용시 타입 오류
- Instanceof 연산자를 이용하여, 먼저 객체 타입을 확인 후 Cast연산해야 함

# Exception 확인하기

API Document에서 해당 클래스에 대한 생성자나 메소드를 검색하면, 그 메소드가 어떠한 Exception을 발생시킬 가능성이 있는지 확인할 수 있다. 해당 메소드를 사용하려면 반드시 뒤에 명시된 예외 클래스를 처리해야 함.

The screenshot shows the Oracle Java API documentation for the `readLine()` method in the `Reader` class. The browser address bar shows `docs.oracle.com/javase/8/docs/api/`. On the left, a sidebar lists various Java packages and classes, with `java.io` selected. The main content area displays the `readLine()` method signature: `public String readLine() throws IOException`. Below the signature, the text explains that the method reads a line of text, terminated by a line feed, carriage return, or carriage return followed by a linefeed. It also lists the `IOException` that is thrown if an I/O error occurs. The `skip` method is also visible below.

the given array. Thus redundant `BufferedReader`s will not copy data unnecessarily.

**Specified by:**  
read in class `Reader`

**Parameters:**  
cbuf - Destination buffer  
off - Offset at which to start storing characters  
len - Maximum number of characters to read

**Returns:**

**readLine**

`public String readLine()  
throws IOException`

Reads a line of text. A line is considered to be terminated by any one of a line feed ('`\n`'), a carriage return ('`\r`'), or a carriage return followed immediately by a linefeed.

**Returns:**  
A String containing the contents of the line, not including any line-termination characters, or null if the end of the stream has been reached

**Throws:**  
`IOException` - If an I/O error occurs

**skip**

`public long skip(long n)  
throws IOException`

Skips characters.

**Overrides:**  
skip in class `Reader`

**Parameters:**  
n - The number of characters to skip



## 1. Exception 처리를 호출한 메소드에게 위임

- 메소드 선언시 throws ExceptionName 문을 추가하여 호출한 상위 메소드에게 처리를 위임하여 해결한다.
- 계속적으로 위임하면 main()까지 위임하게 되고, main()까지 가더라도 예외처리가 되지 않는 경우 JVM이 비정상 종료된다.

## 2. Exception을 발생한 곳에서 직접 처리

- try~catch문을 이용하여 예외를 처리한다.
- try : exception 발생할 가능성이 있는 코드를 try구문 안에 기술한다.
- catch : try 구문에서 exception 발생시 해당하는 exception에 대한 처리를 기술한다.  
여러 개의 exception처리가 가능하나, exception간의 상속관계를 고려해야 한다.
- Finally : exception 발생 여부에 관계 없이, 꼭 처리해야 하는 logic은 finally 안에서 구현한다.  
중간에 return문을 만나도 finally구문은 실행한다.  
단, System.exit(0)를 만나면 무조건 프로그램을 종료한다.  
주로 java.io, java.sql 패키지의 메소드 처리시 이용한다.

# 예외처리 1(Exception Handling)

## throws로 예외 던지기

```
public static void main(String[] args){  
    ThrowTest t = new ThrowTest();  
    try{  
        t.methodA();  
        System.out.println("정상수행");  
    }catch(IOException e){  
        System.out.println("IOException이 발생");  
    }finally{  
        System.out.println("프로그램 종료");  
    }  
}
```

```
public void methodA() throws IOException{  
    methodB();  
}
```

```
public void methodB() throws IOException{  
    methodC();  
}
```

```
public void methodC() throws IOException{  
    throw new IOException();  
    //Exception 발생시키는 구문  
}
```

원칙적으로, 메소드를 호출한 곳에서 예외처리를 해야하므로, 메소드를 최초 호출한 main메소드에서 처리하고 있다.

# 예외처리2(Exception Handling)

## try~catch 표현식

```
try{  
    //반드시 예외 처리를 해야 하는 구문 작성함  
}catch(처리해야할예외클래스명 참조형 변수명){  
    //잡은 예외 클래스에 대한 처리 구문 작성함  
}finally{  
    //실행 도중 해당 Exception이 발생을 하던,  
    //안하던 반드시 실행해야 하는 구문 작성함  
}
```

# try~catch

## try~catch 구문 예시

```
public void test3() {  
    try{  
        System.out.println(100/0); //예외유발코드  
    } catch(ArithmeticException e){  
        e.printStackTrace();  
    }  
}
```

# 예외처리3(Exception Handling)

## 멀티catch 표현식

```
try{  
    //반드시 예외 처리를 해야 하는 구문 작성함  
}catch(예외클래스명3 e){  
  
} catch(예외클래스명2 e){  
  
} catch(예외클래스명1 e){  
  
}
```

이때, catch절의 순서는 상속관계를 따라 작성해야 한다.  
후손클래스가 부모클래스보다 먼저 기술되어야 한다.

FileNotFoundException → IOException → Exception

# try~with~resource 구문

자바 7에서 추가된 기능으로, finally에서 작성되었던 close()처리를 생각하고 자동으로 close처리 되게 하는 문장이다.

## try~with~resource 표현식

```
try(반드시 close 처리 해야 하는 객체에 대한 생성 구문){  
    //예외 처리를 해야 하는 구문 작성함  
}catch(처리해야할예외클래스명 레퍼런스){  
    //잡은 예외 클래스에 대한 처리 구문 작성함  
}finally{  
    //실행 도중 해당 Exception이 발생을 하던,  
    //안하던 반드시 실행해야 하는 구문 작성함  
}
```

# 오버라이딩과 Exception

오버라이딩 시 throws 하는 Exception은 같거나, 더 구체적인 것(후손)이어야 한다.

Object

Exception

SystemException

**ArgumentException**

ArgumentNullException

ArgumentOutOfRangeException

InvalidEnumArgumentException

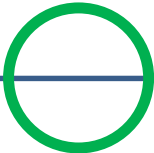
DuplicateWaitObjectException

API Document 참조


# 오버라이딩과 Exception

```
public class TestA{  
    public void methodA() throws IOException{  
        ...  
    }  
}
```

```
public class TestB extends TestA{  
    public void methodA()  
        throws EOFException{  
        ...  
    }  
}
```



```
public class TestC extends TestA{  
    public void methodA()  
        throws Exception{  
        ...  
    }  
}
```





# 사용자 정의 예외생성

Exception 클래스를 상속받아 예외 클래스 작성함.

Exception 발생하는 곳에서 throw new MyException()으로 발생

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

```
public class MyExceptionTest{  
    public void checkAge(int age) throws  
MyException {  
        if(age < 19)  
            throw new MyException("입장불가");  
        else  
            System.out.println("즐감"); }  
}
```

```
public class Run{  
    public static void main(String[] args){  
        MyExceptionTest m  
            = new MyExceptionTest();  
  
        try {  
            m.checkAge(15);  
        }catch(MyException e){  
            System.out.println(e.getMessage());  
        }  
    }  
}
```