

# JAVA 입문 : 이론과 실습



## 제 9장 예외와 단정



# 목차

## ■ 예외

- 예외 정의
- 예외 발생
- 예외 처리
- 예외 전파

## ■ 단정

- 단정의 선언
- 단정 조건 검사 옵션





## 개요 [1/2]

### ■ 예외(exception)

- 실행 시간에 발생하는 에러(run-time error)
- 프로그램의 비정상적인 종료
- 잘못된 실행 결과

compile-time error  
run-time error  
logic error

### ■ 예외 처리(exception handling)

- 기대되지 않은 상황에 대해 예외를 발생
- 야기된 예외를 적절히 처리(exception handler)



## 개요 [2/2]

### ■ 단정(assertion)

- 프로그램이 올바르게 실행되는데 필요한 조건을 선언할 수 있는 언어 기능

### ■ 예외 처리를 언어 시스템에서 제공

- 응용 프로그램의 신뢰성(reliability)을 높임.
- 예외 검사와 처리를 위한 프로그램 코드를 소스에 깔끔하게 삽입

예외처리

PL/I  
Ada  
C++



## 예외 정의 [1/2]

- 예외도 하나의 객체로 취급
  - 따라서, 먼저 예외를 위한 클래스를 정의해야 함.
- 예외 클래스
  - 모든 예외는 형(type)이 **Throwable** 클래스 또는 **그의 서브클래스들** 중에 하나로부터 확장된 클래스의 객체
  - 일반적으로 프로그래머는 Throwable의 서브클래스인 **Exception**을 확장하여 새로운 예외 클래스를 만들어 사용

---

```
class UserErr extends Exception { }  
class UserClass {  
    UserErr x = new UserErr();  
    // ...  
    if (val < 1) throw x;  
}
```

---



## 예외 정의 [2/2]

- 예외에 관련된 메시지를 스트링 형태로 예외 객체에 담아 전달

---

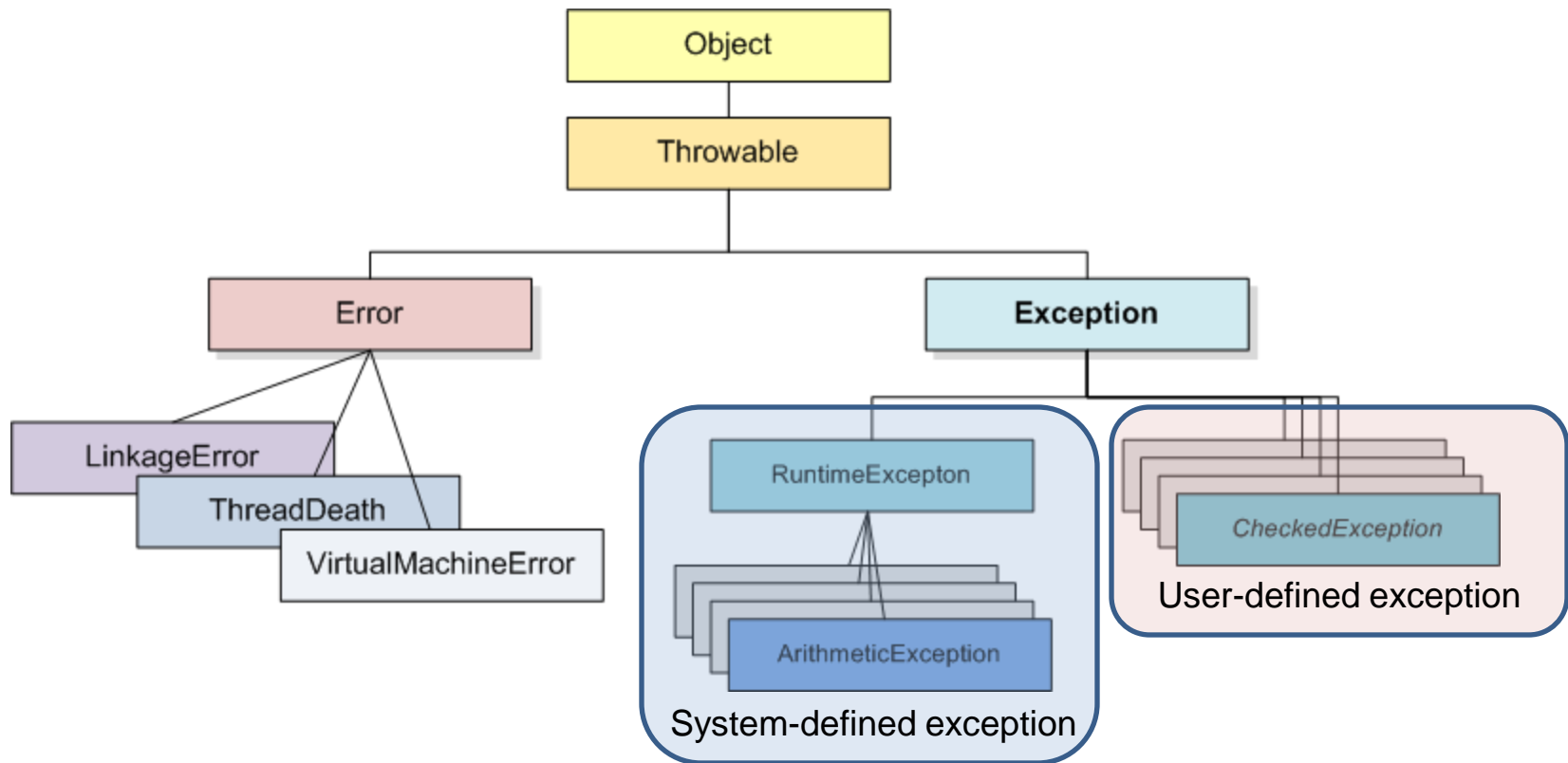
```
class UserErr extends Exception {  
    UserErr(String s) { super(s); }           // constructor  
}  
class UserClass {  
    // ...  
    if (val < 1) throw new UserErr("user exception throw message");  
}
```

---

✦ 예외 처리기에서, `System.out.println(x.getMessage());`



# Throwable 클래스의 계층적 구조



✦ Throwable 클래스에는 예외가 일어난 상황을 설명하는 메시지들을 포함하고 있다.





## 예외의 종류 [1/5]

### ■ 예외의 종류

- System-defined exception(predefined exception)
  - Error 클래스, RuntimeException 클래스
- Programmer-defined exception

### ■ Error 클래스

- 정상적인 응용 프로그램에서는 수용할 수 없는 심각한 에러


### ■ Exception 클래스

- 정상적인 응용 프로그램의 실행에서 발생 가능한 예외를 의미
- 프로그래머에 의해 처리 가능





## 예외의 종류 [2/5]

- 시스템 정의 예외(system-defined exception)
  - 프로그램의 부당한 실행으로 인하여 시스템에 의해 묵시적으로 발생하는 예외
  - Error와 RuntimeException 클래스로부터 확장된 예외
  - 더 이상 프로그램의 실행을 지속할 수 없을 때 자바 시스템에 의해 자동적으로 생성
  - 야기된 예외에 대한 예외 처리기의 유무를 컴파일러가 검사하지 않음  
     **unchecked exception**
- 시스템 예외의 종류
  - ArithmeticException, IndexOutOfBoundsException, NegativeArraySizeException, ...



## 예외의 종류 [3/5]

- **IndexOutOfBoundsException :**
  - 배열, 스트링, 벡터 등과 같이 인덱스를 사용하는 객체에서 인덱스의 범위가 벗어날 때 발생
- **ArrayStoreException :**
  - 배열의 원소에 잘못된 형의 객체를 배정하였을 때 발생
- **NegativeArraySizeException :**
  - 배열의 크기를 음수로 지정하였을 때 발생
- **NullPointerException :**
  - null을 사용하여 객체를 참조할 때 발생
- **SecurityException :**
  - 보안을 위반했을 때 **보안 관리자**(security manager)에 의해 발생
- **IllegalMonitorStateException :**
  - **모니터**(monitor)의 소유자가 아닌 스레드가 wait 또는 notify 메소드를 호출했을 때 발생



Applet 또는  
RMI



## 예외의 종류 [4/6]

- 프로그래머 정의 예외
  - 프로그래머가 필요에 의해 정의
  - Exception 클래스의 서브 클래스
  - 프로그래머에 의해 의도적으로 발생
  - 발생한 예외에 대한 예외 처리기가 존재하는지 컴파일러에 의해 검사, 예외처리가 없으면 에러
    - **checked exception**

---

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

---



## 예외의 종류 [5/5]

### ■ 프로그래머 정의 예외의 예제 프로그램

#### [UserDefinedException.java]

```
class UserErr extends Exception {  
    UserErr(String s) { super(s); }    // constructor  
}  
  
class UserDefinedException {  
    public static void tryException (int val) throws UserErr {  
        if (val < 1) throw new UserErr("user exception throw message");  
    }  
    public static void main(String[] args) {  
        try {  
            System.out.println("try user exception...");  
            tryException(0);  
        } catch (UserErr e) {  
            System.out.println( e.getMessage() );  
        }  
    }  
}
```

실행 결과 :

```
try user exception...  
user exception throw message
```



## 예외 발생 [1/3]

### ■ 예외 발생

- 시스템 정의예외
  - 시스템에 의해 묵시적으로 발생
- 프로그래머 정의 예외
  - 프로그래머가 명시적으로 발생

### ■ throw 구문

- 시스템 정의 예외나 프로그래머 정의 예외를 명시적으로 발생시키는 구문
- 구문 형태 :

```
throw ThrowableObject;
```

Throwable 클래스 혹은  
그의 서브 클래스



## 예외 발생 [2/3]

### ■ throw 를 이용한 예외 발생 예

#### [예제 9.3 - ThrowStatement.java]

```
public class ThrowStatement extends Exception {  
    public static void exp(int ptr) {  
        if (ptr == 0)  
            throw new NullPointerException();  
    }  
    public static void main(String[] args) {  
        int i = 0;  
        ThrowStatement.exp(i);  
    }  
}
```

실행 결과 :

```
java.lang.NullPointerException  
at ThrowStatement.exp(ThrowStatement.java:4)  
at ThrowStatement.main(ThrowStatement.java:8)
```



## 예외 발생 [3/3]

### ■ throws 절

- 프로그래머 정의 예외가 발생하는 경우, 예외 처리기를 갖고 있지 않으면 메소드 선언부분에 명시한다.
- 선언 형태 :

```
modifiers_and_returntype methodName(params) throws e1, ..., ek { }
```

- 명시해 주는 이유는 메소드가 정상적인 복귀 외에 예외에 의해 복귀할 수 있다는 것을 알려 주는 것이다.
- 시스템 정의 예외는 명시해 주지 않는다.





## 예외 처리 [1/3]

### ■ try-catch-finally 구문

- 예외를 검사하고 처리해 주는 문장
- 구문 형태 :

```
try {  
    // ...  
} catch (ExceptionType1 identifier) {  
    // ...  
} catch (ExceptionType2 identifier) {  
    // ...  
} finally {  
    // ...  
}
```

"try 블록"

"catch 블록"

"catch 블록"

"finally 블록"

- try 블록 : 예외 검사되는 블록
- catch 블록: 예외가 처리되는 블록



## 예외 처리 [2/3]

### ■ 예외 처리기의 실행 순서

1. try 블록 내에서 예외가 검사되고 또는 명시적으로 예외가 발생하면,
2. 해당하는 catch 블록을 찾아 처리하고,
3. 마지막으로 finally 블록을 실행한다.

### ■ Default 예외 처리기

- 시스템 정의 예외가 발생했는데도 불구하고 프로그래머가 처리하지 않을 때 처리됨
- 단순히 에러에 대한 메시지를 출력하고 프로그램을 종료하는 기능



## 예외 처리 [3/3]

### [예제 9.8- FinallyClause.java]

```
public class FinallyClause {
    static int count = 0;
    public static void main(String[] args) {
        while (true) {
            try {
                if (++count == 1) throw new Exception();
                if (count == 3) break;
                System.out.println(count + " No exception");
            } catch (Exception e) {
                System.out.println(count + " Exception thrown");
            } finally {
                System.out.println(count + " in finally clause");
            }
        } // end while
        System.out.println("Main program ends");
    }
}
```

실행 결과 :

- 1) Exception thrown
- 1) in finally clause
- 2) No exception
- 2) in finally clause
- 3) in finally clause
- Main program ends

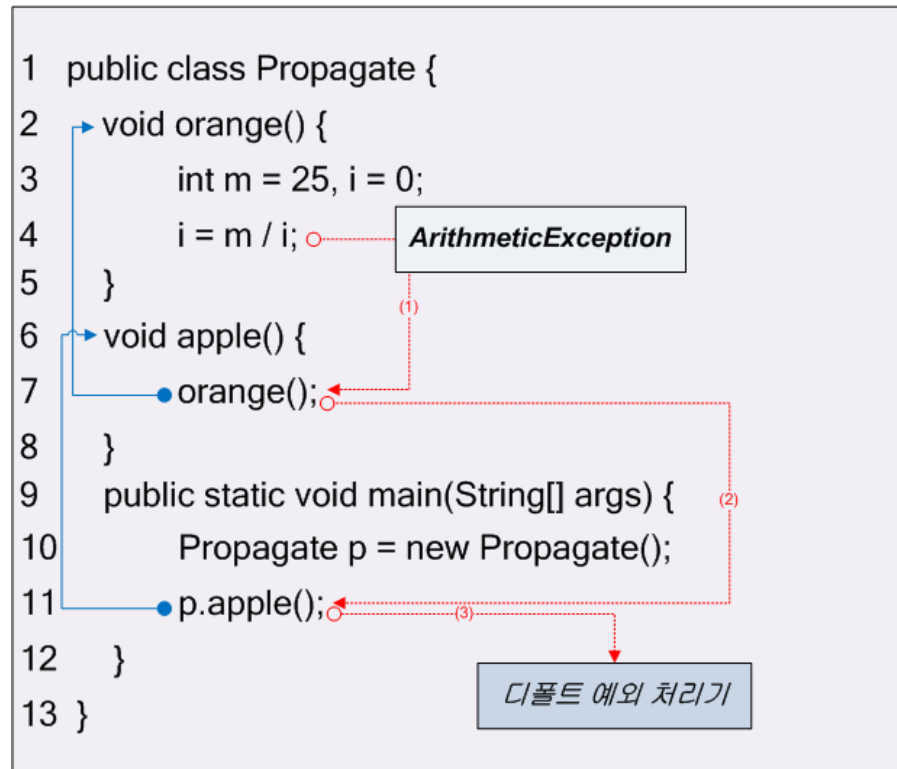


## 예외 전파 [1/3]

- 호출한 메소드로 예외를 전파 (propagation) 하여 특정 메소드에서 모아 처리
  - 예외 처리 코드의 분산을 막을 수 있음
- 예외 전파 순서
  - 예외를 처리하는 catch 블록이 없으면, 호출한 메소드로 예외를 전파
  - 예외 처리기를 찾을 때까지의 모든 실행은 무시



## 예외 전파 [2/3]



```
java.lang.ArithmeticException: / by zero
    at Propagate.orange(Propagate.java:4)
    at Propagate.apple(Propagate.java:7)
    at Propagate.main(Propagate.java:11)
```



## 예외 전파 [3/3]

- 예외 발생 가능성에 대한 명시
  - 시스템 정의 예외
    - 예외의 발생 가능성을 알릴 필요 없음
  - 프로그래머 정의 예외
    - 해당 메소드에서 처리하지 않을 경우, 예외의 종류를 알려야 함
    - throws 절 사용

---

```
public void methodA() throws MyException {  
    // ...  
    if (someErrCondition())  
        throw new MyException();  
    // ...  
}
```

---



## 단정의 선언 [1/3]

### ■ 단정

- 프로그램이 올바르게 실행되는데 필요한 조건을 선언하는 프로그래밍 언어의 기능

### ■ 단정 선언 방법

1. 단정 조건 명시
  2. 단정 조건 명시 + 문자열 정보
- 형식

```
assert <조건식> [: <문자열 정보>];
```

### ■ 단정 조건

- 참이나 거짓의 결과가 되는 조건식
- 참 : 실행이 계속됨
- 거짓 : AssertionError 예외 발생





## 단정의 선언 [2/3]

### ■ 단정의 사용 예

#### [예제 9.13 - AssertExample.java]

```
public class AssertExample {  
    static void drawBox(int x, int y, int w, int h) {  
        assert x >= 0;  
        assert y >= 0;  
        assert w >= 0;  
        assert h >= 0;  
  
        // draw the box.  
    }  
  
    public static void main(String[] args) {  
        drawBox(100, 200, 10, 20);  
        drawBox(0, -10, 5, 30);  
    }  
}
```

실행 결과 :

```
Exception in thread "main" java.lang.AssertionError  
    at AssertWithStringExample.drawBox(AssertExample.java:4)  
    at AssertWithStringExample.main(AssertExample.java:13)
```



## 단정의 선언 [3/3]

### ■ 문자열 정보가 추가된 단정의 사용 예

#### [예제 9.13 - AssertWithStringExample.java]

```
public class AssertWithStringExample {  
    static void drawBox(int x, int y, int w, int h) {  
        assert (x >= 0) : "x must be 0 or more.";  
        assert (y >= 0) : "y must be 0 or more.";  
        assert (w >= 0) : "w must be 0 or more.";  
        assert (h >= 0) : "h must be 0 or more.";  
  
        // draw the box.  
    }  
  
    public static void main(String[] args) {  
        drawBox(100, 200, 10, 20);  
        drawBox(0, -10, 5, 30);  
    }  
}
```

실행 결과 :

Exception in thread "main" java.lang.AssertionError: y must be 0 or more.  
at AssertWithStringExample.drawBox(AssertWithStringExample.java:4)  
at AssertWithStringExample.main(AssertWithStringExample.java:13)



## 단정 조건 검사 옵션

### ■ 자바의 기본설정

#### ■ 단정 검사를 하지 않음

- 단정에 명시된 조건 검사는 실행 속도를 느리게 함

### ■ 단정 조건 검사 설정

#### ■ 단정 조건 검사

```
<jdk_path>/bin/java -ea <실행할 클래스 이름>  
<jdk_path>/bin/java -enableassertions <실행할 클래스 이름>
```

#### ■ 단정 조건 무시

```
<jdk_path>/bin/java -da <실행할 클래스 이름>  
<jdk_path>/bin/java -disableassertions <실행할 클래스 이름>
```



# 단정과 예외처리의 차이점

## ■ 유사점

- 자바 프로그램의 신뢰성 향상을 위해 사용
- 실행 중에 문제가 생기면 예외 발생

## ■ 차이점

### ■ 단정

- 실행에 필요한 조건을 검사
- 단정은 자바가상기계의 실행 옵션에 따라 검사 생략이 가능

### ■ 예외

- 프로그램 상에서 발생하는 예기치 않은 구문들을 처리
- 항상 예외처리 구문을 수행



## 단원 요약 [1/2]

### ■ 예외 처리 목적

- 별개의 통로를 제공하여 더욱 안전한 프로그램을 작성

### ■ 예외 처리의 상황

- 에러를 수정하고 예외를 발생시킨 메소드의 재호출이 필요한 경우
- 메소드의 재호출 없이 에러를 수정하고 실행을 계속하는 경우
- 메소드가 그의 실행 결과를 포기하는 대신에 대안적인 결과가 필요한 경우
- 발생한 예외를 적절히 처리한 후, 호출자에게 동일 예외 또는 다른 예외를 재 발생시킬 필요가 있는 경우
- 예외가 일어 났을 때 프로그램을 종료하려는 경우



## 단원 요약 [2/2]

- 단정 목적
  - 프로그램의 실행 조건을 검사하여 견고한 프로그램을 작성
- 단정 사용 상황
  - 실행 조건의 기술을 통한 검사의 필요 시
  - 예외처리와 달리 테스트 과정에서만 검사를 필요로 하는 경우