● 메서드에 예외 선언하기: 메서드 호출 시 발생 가능한 예외를 메서드를 호출하는 쪽에 알리는 것. Exception클래스와 그 자손들에 해당하는 반드시 처리해야 하는 예외(체크드 예외)는 메서드 선언부에 선언하여 메서드를 호출하는 메서드에서 처리하도록 예외를 떠넘길 수 있다. 이렇게 하므로써 메서드를 사용하려는 사람이 메서드의 선언부를 보았을 때, 이 메서드를 사용하기 위해서는 어떤 예외들이 반드시 처리되어야 하는지 쉽게 알 수 있다. Java API문서를 통해 사용하고자 하는 메서드의 선언부와 throws:를 보고, 이 메서드에는 어떤 예외가 발생할 수 있으며 반드시 처리해야주어야 하는 예외는 어떤 것들이 있는지 확인하는 것이 좋다. 키워드 throws를 사용해서 메서드 선언부에 선언하며, 예외가 여러 개일 경우에는 쉼표로 구분한다. 만일 모든 예외의 최고 조상인 Exception클래스를 메서드에 선언하면, 이 메서드는 모든 종류의 예외가 발생할 가능성이 있다는 뜻이다. 메서드를 오버라이딩 할 때는 단순히 선언된 예외의 개수가 아니라 상속관계까지 고려해야 한다. 오바라이딩의 규칙은 1) 메서드의 선언부가 동일해야 하고 2) 접근제어자가 더 좁은 범위이면 안 되며 3) 조상보다 더 많은 예외를 선언하면 안 된다.

```
void method() throws Exception1, Exception, ... ExceptionN {
    // 메서드 내용
}
```

예외를 선언한 메서드를 호출한 메서드에게 예외를 전달하여 예외처리를 떠맡길 수도 있다. 예외를 전달받은 메서드가 또다시 자신을 호출한 메서드에게 전달할 수 있으며, 이런 식으로 계속 호출스택에 있는 메서드들을 따라 전달되다가 제일 마지막에 있는 main메서드에서도 예외가 처리되지 않으면, main메서드마저 종료되어 프로그램이 예외로 인해 비정상적으로 종료된다. 결국 어느 한 곳에서는 반드시 try-catch문으로 예외처리를 해 주어야 한다.

메서드에 선언된 예외는 1) 예외가 발생한 메서드 내에서 처리하거나, 2) 예외가 발생한 메서드를 호출한 메서드(main method)에서 처리할 수도 있고 — 이 경우 throws을 사용해 메서드에 예외를 선언하여 예외를 떠넘긴다, 3) 또는 두 메서드가 예외처리를 분담할 수도 있다—몇 개는 try-catch문을 통해서 메서드 내에서 자체적으로 처리하고, 그 나머지는 선언부에 지정하여 호출한 메서드에서 처리하는 방식. 예외가 발생한 메서드 내에서 자체적으로 처리해도 되는 것은 메서드 내에서 try-catch문을 사용해서 처리하고, 메서드 내에서 자체적으로 해결이 안 되는 경우(ex. 메서드에 호출 시 넘겨받아야 할 값을 다시 받아야 하는 경우)에는 예외를 메서드에 선언해서, 호출한 메서드에서 처리해야 한다.

```java
// Ex. 1) 예외가 발생한 메서드 내에서 예외 처리

import java.io.*;

class ExceptionEx15 {
    public static void main(String[] args) {
        // command line에서 입력받은 값을 이름으로 갖는 파일을 생성한다.
        File f = createFile(args[0]);
        System.out.println( f.getName() + " 파일이 성공적으로 생성되었습니다.");
    } // main메서드의 끝

    static File createFile(String fileName) {
        try {
            if (fileName==null || fileName.equals(""))
                throw new Exception("파일이름이 유효하지 않습니다.");
        } catch (Exception e) {
             // fileName이 부적절한 경우, 파일 이름을 '제목없음.txt'로 한다.
            fileName = "제목없음.txt";
        } finally {
            File f = new File(fileName); // File클래스의 객체를 만든다.
            createNewFile(f);            // 생성된 객체를 이용해서 파일을 생성한다.
            return f;
        }
    }   // createFile메서드의 끝

    static void createNewFile(File f) {
        try {
            f.createNewFile();      // 파일을 생성한다.
        } catch(Exception e){ }
    }   // createNewFile메서드의 끝
}
```

```java
// Ex. 2) 예외가 발생한 메서드를 호출한 메서드에서 예외 처리

import java.io.*;

class ExceptionEx16 {
    public static void main(String[] args) {
        try {
            File f = createFile(args[0]);
            System.out.println( f.getName()+"파일이 성공적으로 생성되었습니다.");
        } catch (Exception e) {
            System.out.println(e.getMessage()+" 다시 입력해 주시기 바랍니다.");
        }
    }   // main메서드의 끝

    static File createFile(String fileName) throws Exception {
        if (fileName==null || fileName.equals(""))
            throw new Exception("파일이름이 유효하지 않습니다.");
        File f = new File(fileName);        //  File클래스의 객체를 만든다.
        // File객체의 createNewFile메서드를 이용해서 실제 파일을 생성한다.
        f.createNewFile();
        return f;       // 생성된 객체의 참조를 반환한다.
    }   // createFile메서드의 끝
}   // 클래스의 끝
```

● finally 블럭: 예외의 발생여부에 상관없이 실행되어야 할 코드를 포함시킬 목적으로 사용된다. 예외가 발생한 경우에는 'try-catch-finally' 순서로, 예외가 발생하지 않은 경우 'try-finally'순으로 실행된다. try블럭이나 catch블럭에서 return문을 만나 실행되는 경우에도 finally블럭의 문장들이 먼저 실행된 후에 현재 실행 중인 메서드를 종료한다.

```
try {
    // 예외가 발생할 가능성이 있는 문장들을 넣는다.
} catch (Exception1 e1) {
    // 예외처리를 위한 문장을 넣는다.
} finally {
    // 예외의 발생여부에 관계없이 항상 수행되어야 하는 문장들을 넣는다.
    // finally블럭은 try-catch문의 맨 마지막에 위치해야 한다.
}
```

Resources: 자바의 정석