

Prefer try-with-resources to try-finally

개요

java library들은 close() method를 호출하여 수동적으로 반드시 닫아줘야하는 자원들이 많습니다.

예시로 InputStream, OutputStream, java.sql.Connection 같은 것들이 있지요. (cf. 이 코드들은 안전망으로 finalizer가 있습니다.)

전통적인 방법의 문제점

자원을 닫는 코드는 보기에 지저분할 수 있습니다. 전통적으로 쓰였던 try-finally구문을 보죠.

예제

1. try - one

```
public class TopLine {
    // try-finally - No longer the best way to close resources! (page 34)
    static String firstLineOfFile(String path) throws IOException {
        BufferedReader br = new BufferedReader(new FileReader(path));
        try {
            return br.readLine();
        } finally {
            br.close();
        }
    }

    public static void main(String[] args) throws IOException {
        String path = args[0];
        System.out.println(firstLineOfFile(path));
    }
}
```

이 코드는 보기에 좋을 수도 있습니다... 하지만 아래 코드를 보면 머리 아플겁니다.

2. try - two

```
public class Copy {
    private static final int BUFFER_SIZE = 8 * 1024;

    // try-finally is ugly when used with more than one resource! (Page 34)
    static void copy(String src, String dst) throws IOException {
        InputStream in = new FileInputStream(src); //fault
        try {
            OutputStream out = new FileOutputStream(dst);
            try {
                byte[] buf = new byte[BUFFER_SIZE];
                int n;
                while ((n = in.read(buf)) >= 0)
                    out.write(buf, 0, n);
            } finally {
                out.close();
            }
        } finally {
            in.close();
        }
    }

    public static void main(String[] args) throws IOException {
        String src = args[0];
        String dst = args[1];
        copy(src, dst);
    }
}
```

읽기도 어려울 뿐더러, 실수까지 발생할 수 있습니다. `InputStream in = new FileInputStream(src);` 이 부분이 try 안에 선언되지 않아서 try-catch에서 예외를 잡을 수 없죠.

예외 처리 문제

try - finally에서, try에서 exception이 발생하여 finally를 실행했다고 봅시다. finally에서도 exception이 발생한다면 try block에서 발생한 Exception 사유는 제거되고 finally의 exception으로 덮어 씌워집니다.

이는 추후 디버깅을 어렵게 만드는 요소이지요.

따라서 Java 7에선 아래 내용을 추가하였습니다.

try-with-resources

사용법

우선, 자원이 `AutoCloseable` interface를 구현해야 합니다. 이는 단순히 `void close()` method만 선언되어 있습니다.

```
public interface AutoCloseable {  
    /**  
     * Closes this resource, relinquishing any underlying resources.  
     * This method is invoked automatically on objects managed by the  
     * {@code try}-with-resources statement.  
     *  
     * <p>While this interface method is declared to throw {@code  
     * Exception}, implementers are <em>strongly</em> encouraged to  
     * declare concrete implementations of the {@code close} method to  
     * throw more specific exceptions, or to throw no exception at all  
     * if the close operation cannot fail.  
     *  
     * <p>Cases where the close operation may fail require careful  
     * attention by implementers. It is strongly advised to relinquish  
     * the underlying resources and to internally <em>mark</em> the  
     * resource as closed, prior to throwing the exception. The {@code  
     * close} method is unlikely to be invoked more than once and so  
     * this ensures that the resources are released in a timely manner.  
     * Furthermore it reduces problems that could arise when the resource  
     * wraps, or is wrapped, by another resource.  
     *  
     * <p><em>Implementers of this interface are also strongly advised  
     * to not have the {@code close} method throw {@link  
     * InterruptedException}.</em>  
     *  
     * This exception interacts with a thread's interrupted status,  
     * and runtime misbehavior is likely to occur if an {@code  
     * InterruptedException} is {@linkplain Throwable#addSuppressed  
     * suppressed}.  
     *  
     * More generally, if it would cause problems for an  
     * exception to be suppressed, the {@code AutoCloseable.close}  
     * method should not throw it.  
     *  
     * <p>Note that unlike the {@link java.io.Closeable#close close}  
     * method of {@link java.io.Closeable}, this {@code close} method  
     * is <em>not</em> required to be idempotent. In other words,  
     * calling this {@code close} method more than once may have some  
     * visible side effect, unlike {@code Closeable.close} which is  
     * required to have no effect if called more than once.
```

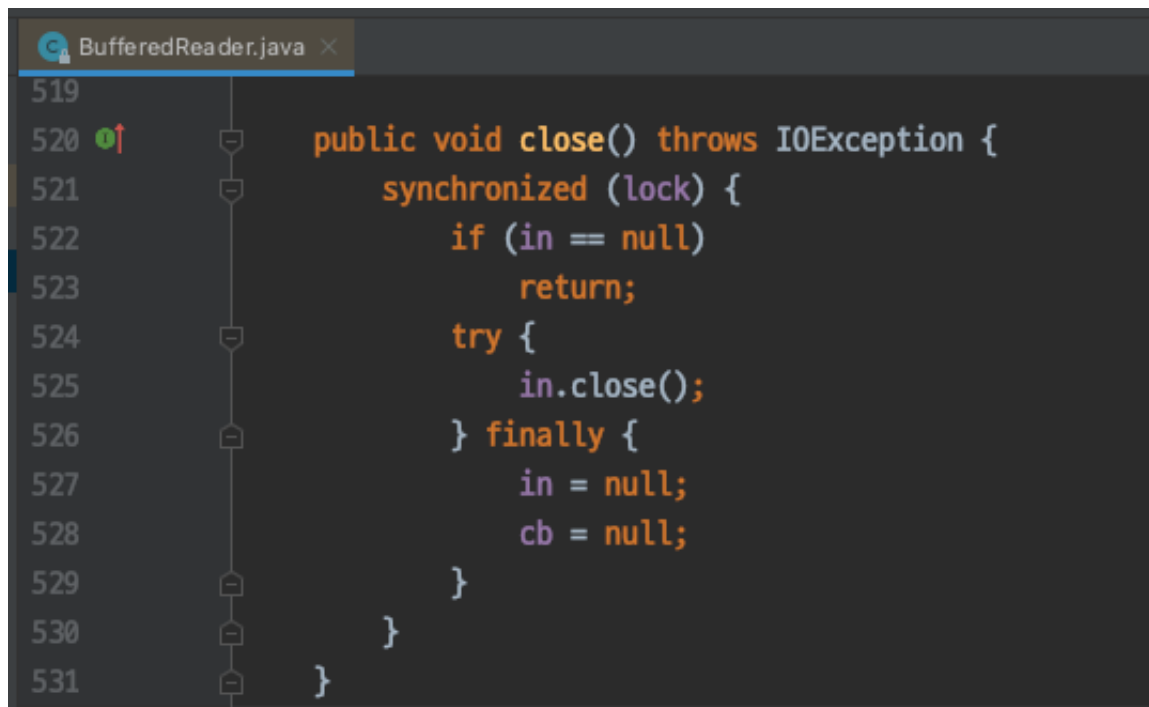
```

*
* However, implementers of this interface are strongly encouraged
* to make their {@code close} methods idempotent.
*
* @throws Exception if this resource cannot be closed
*/
void close() throws Exception;
}

```

대부분 java 라이브러리들은 AutoCloseable을 구현했기 때문에 try with resources에 적용할 수 있습니다.

하단은 BufferedReader에서 구현한 코드입니다.



```

BufferedReader.java x
519
520 public void close() throws IOException {
521     synchronized (lock) {
522         if (in == null)
523             return;
524         try {
525             in.close();
526         } finally {
527             in = null;
528             cb = null;
529         }
530     }
531 }

```

try-with-resource syntax (JLS 8)

14.20.3 try-with-resources

A `try-with-resources` statement is parameterized with variables (known as *resources*) that are initialized before execution of the `try` block and closed automatically, in the reverse order from which they were initialized, after execution of the `try` block. `catch` clauses and a `finally` clause are often unnecessary when resources are closed automatically.

TryWithResourcesStatement:

`try ResourceSpecification Block [Catches] [Finally]`

ResourceSpecification:

`(ResourceList [;])`

ResourceList:

`Resource { ; Resource }`

Resource:

`{VariableModifier} UnannType VariableDeclaratorId = Expression`

See §8.3 for *UnannType*. The following productions from §4.3, §8.3, and §8.4.1 are shown here for convenience:

VariableModifier:

(one of)

`Annotation final`

VariableDeclaratorId:

`Identifier [Dims]`

Dims:

`{Annotation} [] {{Annotation} [] }`

이제 개선된 코드를 보겠습니다.

개선된 코드

1. try-one (enhanced)

```
static String firstLineOfFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(
        new FileReader(path))) {
        return br.readLine();
    }
}
```

2. try-two (enhanced)

```
static String firstLineOfFile(String path, String defaultVal) {
    try (BufferedReader br = new BufferedReader(
        new FileReader(path))) {
        return br.readLine();
    } catch (IOException e) {
        return defaultVal;
    }
}
```

와우.. 딱봐도 짧아지고 읽기 편해졌습니다. 그리고 버그도 찾기 쉬워졌죠.

디버깅

추적

만약에 readLine()과 close() 둘다 exception을 던진다면, close()의 Exception은 readLine()에 매달리게 됩니다. stack trace엔 이는 suspended라는 꼬리표를 달고 출력합니다.

또는 자바 7에서 추가된 Throwable의 getSuppressed메서드를 이용하면 프로그램 코드에서 가져올 수도 있습니다.

catch

try-with-resources에서도 catch절을 쓸 수 있어서 try문을 중첩하지 않고도 다수의 예외를 처리할 수 있습니다. 아래 코드처럼 말이죠.

```
public class Copy {
    private static final int BUFFER_SIZE = 8 * 1024;

    // try-with-resources on multiple resources - short and sweet (Page 35)
    static void copy(String src, String dst) throws IOException {
        try (InputStream in = new FileInputStream(src);
            OutputStream out = new FileOutputStream(dst)) {
            byte[] buf = new byte[BUFFER_SIZE];
            int n;
            while ((n = in.read(buf)) >= 0)
                out.write(buf, 0, n);
        }
    }

    public static void main(String[] args) throws IOException {
        String src = args[0];
    }
}
```

```
String dst = args[1];  
copy(src, dst);  
}  
}
```