● try-with-resources문: 자동 자원 반환. 주로 입출력(I/O)과 관련된 클래스를 사용할 때 유용하다. 입출력 클래스 중에는 사용한 후에 꼭 닫아주어야 사용했던 자원(resources)이 반환되는 것들이 있어서 finally블럭 안에 close()를 넣어주어야 하는 것들이 있다. 하지만 이때 문제는 close()가 예외를 발생시킬 수 있다는 것에 있는데, 이때 finally블럭 안에 try-catch문을 추가해서 close()에서 발생할 수 있는 예외를 처리하도록 변경해버리면 코드가 복잡해질 뿐만이 아니라 try블럭과 finally블럭 안에서 모두 예외가 발생하면, try블럭의 예외는 무시된다. 마지막으로 발생한 예외에 대한 사항만 처리가 되기 때문이다. 그래서 나온 것이 try-with-resources문인데, 괄호()안에 객체를 생성하는 문장을 넣으면, 이 객체는 따로 close()를 호출하지 않아도 try블럭을 벗어나는 순간ㅌ 자동적으로 close()가 호출된다. 그 다음에 catch블럭 또는 finally블럭이 수행된다. try블럭의 괄호()안에는 변수를 선언하는 것도 가능하며, 선언된 변수는 try블럭 내에서만 사용할 수 있다. 이처럼 try-with-resources문에 의해 자동적으로 객체의 close()가 호출될 수 있으려면, 클래스가 AutoCloseable이라는 인터페이스를 구현한 것이어야만 한다.

```
public interface AutoCloseable {
    void close() throws Exception;
}
```

만약 메서드와 close()에서 모두 예외가 발생하면, close()에서 발생한 예외는 억제된(suppressed)라는 의미의 머리말과 함께 출력된다. 두 예외가 동시에 발생할 수는 없기 때문에, 실제 발생한 예외를 메서드로 하고 close()에서 발생한 예외는 억제된 예외로 다둔다. 이때, 억제된 예외에 대한 정보는 실제로 발생한 예외에 저장된다. Throwable에는 억제된 예외와 관련된 다음과 같은 메서드가 정의되어 있다.

```
void addSuppressed(Throwable exception)   // 억제된 예외를 추가
Throwable[] getSupressed()                // 억제된 예외(배열)을 반환
```

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

class CloseException extends Exception {
    CloseException(String msg) { super(msg); }
}

class CloseableResource implements AutoCloseable {
    public void exceptionWork(boolean exception) throws WorkException {
        System.out.println("exceptionWork("+exception+")가 호출됨");

        if(exception)
            throw new WorkException("WorkException발생!!!");
    }

    public void close() throws CloseException {
        System.out.println("close()가 호출됨");
        throw new CloseException("CloseException발생!!!");
    }
}

class TryWithResourceEx {
    public static void main(String args[]) {

        try (CloseableResource cr = new CloseableResource()) {
            cr.exceptionWork(false); // 예외가 발생하지 않는다. exceptionWork(false)가 호출
            // close()가 호출됨
        } catch(WorkException e) {
            e.printStackTrace();
        } catch(CloseException e) {
            e.printStackTrace(); // CloseException: CloseException 발생!! at ...
        }
        System.out.println();

        try (CloseableResource cr = new CloseableResource()) {
            cr.exceptionWork(true); // 예외가 발생한다. exceptionWork(true)가 호출됨
            // close()가 호출됨
        } catch(WorkException e) {
            e.printStackTrace(); // WorkException발생!! at...
        } catch(CloseException e) { //Suppressed: CloseException: CloseException발생!!!!
            e.printStackTrace();
        }
    } // main의 끝
}
```

● 사용자 정의 예외 만들기: 프로그래머가 Exception클래스 또는 RuntimeException클래스로부터 상속 받아 새로운 예외 클래스를 만들 수 있다. 필요하다면, 멤버 변수나 메서드를 추가할 수 있으며, Exception클래스는 생성 시에 String값을 받아서 메시지로 저장할 수 있다.

```
Class MyException extends Exception {
    // 에러 코드 값을 저장하기 위한 필드를 추가했다.
    private final int ERROR_CODE;

    MyException(String msg, int errCode) {        // 생성자
         super(msg);
         ERR_CODE = errCode;
    }

    MyException(String msg) {        // 생성자
        this(msg, 100);       // ERR_CODE를 100(기본값)으로 초기화 한다.
    }

    public int getErrCode() {      // 에러 코드를 얻는 개터 함수
        return ERR_CODE;           // 이 메서드는 주로 getMessage()와 함께 사용될 것이다.
    }
}

// 이렇게 함으로써 MyException이 발생했을 때, catch블럭에서 getMessage()와 getErrCode()를
// 함께 사용해서 에러코드와 메세지를 모두 얻을 수 있다.
```

기존의 예외 클래스는 주로 Exception을 상속받아서 'checked 예외'로 작성하는 경우가 많았지만, 요즘은 예외처리를 선택적으로 할 수 있도록 RuntimeException을 상속받아서 작성하는 쪽으로 바뀌어가고 있다. checked예외는 반드시 예외처리를 해주어야 하기 때문에 예외처리가 불필요한 경웨도 try-catch문을 넣어서 코드가 복잡해지기 때문이다.

● 예외 되던지기(exception re-throwing): 예외를 처리한 후에 인위적으로 다시 발생시키는 방법을 통해서 단 하나의 예외에 대해서 예외가 발생한 메서드와 호출한 메서드, 양쪽에서 처리하도록 하는 것. 먼저 예외가 발생할 가능성이 있는 메서드에서 try-catch문을 사용해서 예외를 처리해 주고, catch문에서 필요한 작업을 행한 후에 throw문을 사용해서 예외를 다시 발생시킨다. 이때 이 메서드의 선언부에 발생할 예외를 throws에 지정해주어, 다시 발생한 예외는 이 메서드를 호출한 메서드에게 전달되고, 호출한 메서드의 try-catch문에서 예외를 도다시 처리한다. 반환값이 있는 return문의 경우, catch블럭에도 return문이 있어야 한다. 예외가 발생했을 경우에도 값을 반환해야 하기 때문이다. 또는 catch블럭에서 예외 되던지기를 해서 호출한 메서드로 예외를 전달하면, return문이 없어도 된다. (finally블럭 내에서도 return문을 사용할 수 있으며, try블럭이나 catch블럭의 return문 다음에 수행되며 최종적으로는 finally블럭 내에 return문의 값이 반환된다).

```
class Ex {
    static int method1() throws Exception {
        try {
            System.out.println("method1()이 호출되었습니다.");
            return 0;
        } catch (Exception e) {
            e.printStackTrace();
            // return 1; catch 블럭 내에도 return문이 필요하다.
            throw e; // 다시 예외를 발생시켜 return문 대신 예외를 호출한 메서드로 전달.
        } finally {
            System.out.println("method1()의 finally블럭이 실행되었습니다.");
        }
    }

    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception e) {
            System.out.println("main메서드에서 예외가 처리되었습니다.");
        }
    }
}
```

● 연결된 예외(chained exception): 한 예외가 다른 예외를 발생시킬 수 있으며, 예외 A가 예외 B를 발생시켰다면, A를 B의 원인 예외(cause exception)이라고 한다. initCause()는 지정한 예외를 원인 예외로 등록하는 메서드로 Exception클래스의 조상인 Throwable클래스에 정의되어 있기 때문에, 모든 예외에서 사용가능하다. initCause()로 원인 예외로 등록하고 나서 throw로 이 예외를 던지는데, 이렇게 하는 이유는 여러가지 예외를 하나의 큰 부류의 예외로 묶어서 다루기 위해서이다. 만약 두 예외의 공통 조상의 예외를 사용해 catch블럭을 작성하면, 실제로 발생한 예외가 두 예외 중 어느 것인지 알 수 없다는 문제가 생기고, 그렇다고 두 예외의 상속관계를 변경해야 한다는 것도 부담이기 때문에 예외가 원인 예외를 포함할 수 있게 한 것이다. 이렇게 하면, 두 예외는 상속관계가 아니어도 상관없다. 또다른 이유는 checked예외를 unchecked예외로 바꿀 수 있도록 하기 위해서이다. Exception클래스의 자손인 예외를 throw new RuntimeException()으로 감싸면 unchecked예외가 되며, 더이상 해당 메서드의 선언부에 예외를 선언하지 않아도 된다 — 이것은 RuntimeException의 원인 예외를 등록하는 생성자를 사용한 것이다.

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

class SpaceException extends Exception {
    SpaceException(String msg) {
        super(msg);
    }
}

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

```
class ChainedExceptionEx {
    static void copyFiles() { /* 파일들을 복사하는 코드를 적는다. */ }

    static void deleteTempFiles() { /* 임시파일들을 삭제하는 코드를 적는다.*/}

    static boolean enoughSpace()   {
        // 설치하는데 필요한 공간이 있는지 확인하는 코드를 적는다.
        return false;
    }

    static boolean enoughMemory() {
        // 설치하는데 필요한 메모리공간이 있는지 확인하는 코드를 적는다.
        return true;
    }

    static void startInstall() throws SpaceException, MemoryException {
        if(!enoughSpace()) {        // 충분한 설치 공간이 없으면...
            throw new SpaceException("설치할 공간이 부족합니다.");
        }

        if (!enoughMemory()) {      // 충분한 메모리가 없으면...
            throw new MemoryException("메모리가 부족합니다.");
//      throw new RuntimeException(new MemoryException("메모리가 부족합니다."));
        }
    } // startInstall메서드의 끝

    static void install() throws InstallException {
        try {
            startInstall();     // 프로그램 설치에 필요한 준비를 한다.
            copyFiles();        // 파일들을 복사한다.
        } catch (SpaceException e)  {
            InstallException ie = new InstallException("설치중 예외발생");
            ie.initCause(e);
            throw ie;
        } catch (MemoryException me) {
            InstallException ie = new InstallException("설치중 예외발생");
            ie.initCause(me);
            throw ie;
        } finally {
            deleteTempFiles();  // 프로그램 설치에 사용된 임시파일들을 삭제한다.
        } // try의 끝
    }

    public static void main(String args[]) {
        try {
            install();
        } catch(InstallException e) {
            e.printStackTrace();
        } catch(Exception e) {
            e.printStackTrace();
        }
    } // main의 끝

} // ExceptionTest클래스의 끝
```

```
InstallException: 설치중 예외발생
    at ChainedExceptionEx.install(ChainedExceptionEx.java:50)
    at ChainedExceptionEx.main(ChainedExceptionEx.java:64)
Caused by: SpaceException: 설치할 공간이 부족합니다.
    at ChainedExceptionEx.startInstall(ChainedExceptionEx.java:36)
    at ChainedExceptionEx.install(ChainedExceptionEx.java:47)
    ... 1 more
```

Resources: 자바의 정석