JAVA 프로그래밍

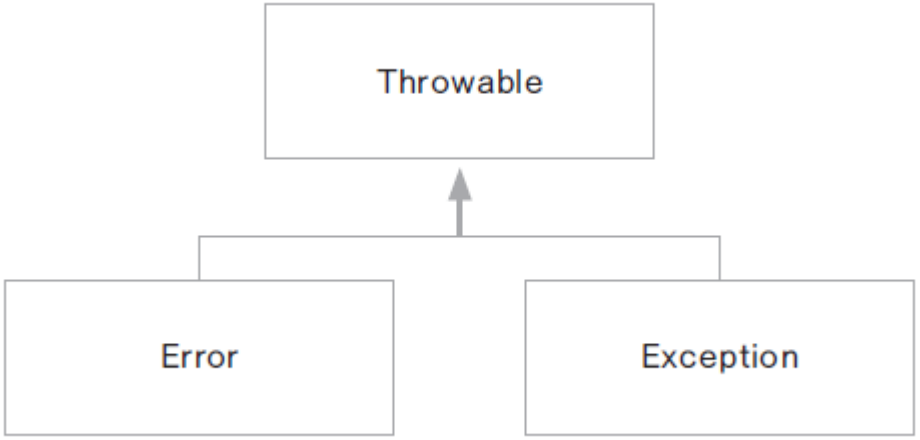
예외 처리

오류란 무엇인가요?

- **컴파일 오류**(compile error)
  : 프로그램 코드 작성 중 발생하는 문법적 오류
- **실행 오류**(runtime error)
  : 실행 중인 프로그램이 의도하지 않은 동작(bug)을 하거나 프로그램이 중지되는 오류
- 실행 오류 시 비정상 종료는 서비스 운영에 치명적
- 오류가 발생할 수 있는 경우, 로그(log)를 남겨 추후 이를 분석하여 오류의 원인을 찾아야 함
- 자바는 예외 처리를 통하여 프로그램의 비정상 종료를 막고 log를 남길 수 있음

오류와 예외 클래스

- **시스템 오류**(error) : 가상 머신(JVM)에서 발생, 프로그래머가 처리 할 수 없음
  동적 메모리가 없는 경우, 스택 오버플로우 등
- **예외**(Exception) : 프로그램에서 제어 할 수 있는 오류
  읽어 들이려는 파일이 존재하지 않는 경우, 네트워크 연결이 끊어진 경우

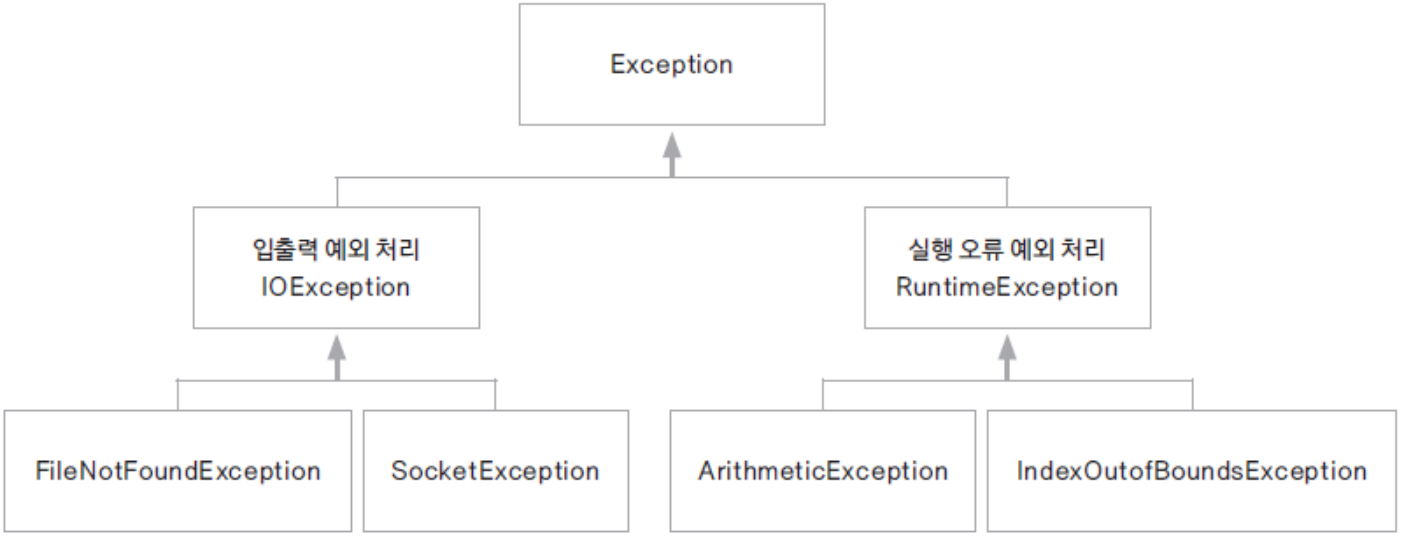


예외 클래스의 종류

- 모든 예외 클래스의 최상위 클래스는 Exception
- 다양한 예외 클래스가 제공 됨

Class Exception

java.lang.Object
    java.lang.Throwable
        java.lang.Exception



예외 처리하기

- try-catch 문

```
try {
   예외가 발생할 수 있는 코드 부분
} catch(처리할 예외 타입 e) {
    try 블록 안에서 예외가 발생했을 때 예외를 처리하는 부분
}
```

- try- catch 문 사용

```
try {
 for(int i = 0; i <= 5; i++) {
  arr[i] = i;
  System.out.println(arr[i]);
 }
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println(e);
    System.out.println("예외 처리 부분");
}
```

예외가 발생할 수 있으므로 try 블록에 작성

예외가 발생하면 catch 블록 수행

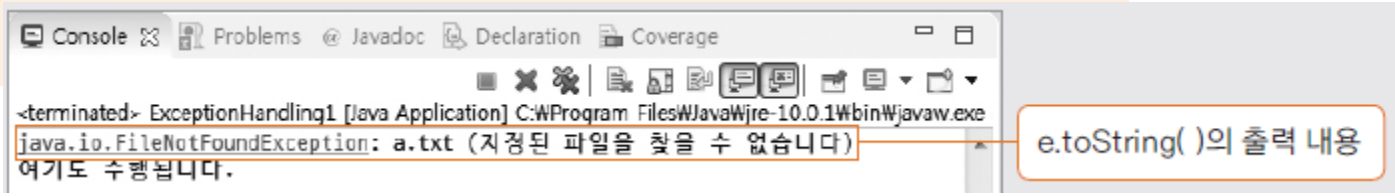
try ~ catch 사용하기

```
public class ArrayExceptionHandling {
    public static void main(String[] args) {
        int[] arr = new int[5];

        try {    // 예외가 발생할 수 있는 코드
            for(int i = 0; i <= 5; i++) {
                arr[i] = i;
                System.out.println(arr[i]);
            }
        } catch(ArrayIndexOutOfBoundsException e) { // 예외발생 시 수행되는 블록
            System.out.println(e);
            System.out.println("예외 처리 부분");
        }
        System.out.println("프로그램 종료");
    }
}
```

try-catch문 예제

```
public class ExceptionHandling1 {
   public static void main(String[ ] args) {
      try {
         FileInputStream fis = new FileInputStream("a.txt");
      } catch (FileNotFoundException e) {
         System.out.println(e);  //예외 클래스의 toString( ) 메서드 호출
      }
      System.out.println("여기도 수행됩니다.");    //정상 출력
   }
}
```



- ☞ 비정상 종료되지 않아 “여기도 수행됩니다.” 부분 출력됨
  - 예외 처리를 하면, 예외 상황을 알려주는 메시지를 볼 수 있고, 프로그램이 비정상 종료되지 않고 계속 수행되도록 만들 수 있음

try-catch-finally 문

- finally 블록에서 프로그램 리소스를 정리함
- try{} 블록이 실행되면 finally{} 블록은 항상 실행됨
- 리소스를 정리하는 코드를 각 블록에서 처리하지 않고 finally에서 처리함

```
try {
    예외가 발생할 수 있는 부분
} catch(처리할 예외 타입 e) {
    예외를 처리하는 부분
} finally {
    항상 수행되는 부분
}
```

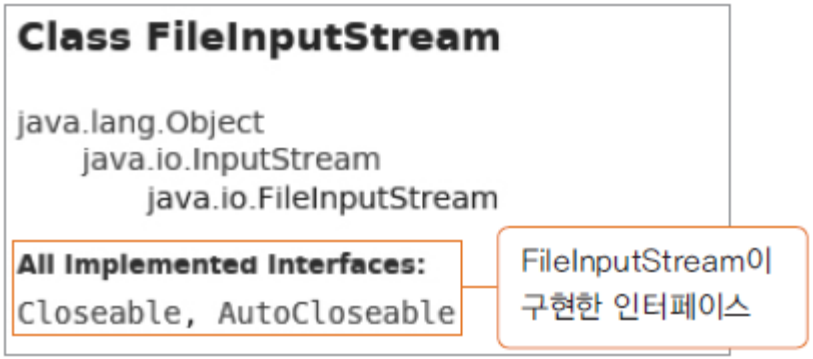
try-catch-finally문 예제

```
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ExceptionHandling2 {
    public static void main(String[] args) {
        FileInputStream fis = null;
        try {
            fis = new FileInputStream("a.txt");
        } catch (FileNotFoundException e) {
            System.out.println(e);
            return;
        } finally {
            if(fis != null) {
                try {
                    fis.close();    // 파일 스트림 닫기
                } catch (IOException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            System.out.println("항상 수행 됩니다.");
        }
        System.out.println("여기도 수행됩니다.");
    }
}
```

try-with-resources문

- 리소스를 자동 해제 하도록 제공해주는 구문
- 자바 7 부터 제공됨
- close()를 명시적으로 호출하지 않아도 try{ } 블록에서 열린 리소스는 정상적인 경우, 예외 발생한 경우 모두 자동 해제됨
- **해당 리소스가 AutoCloseable을 구현 해야 함**
- FileInputStream의 경우 AutoCloseable을 구현하고 있음



AutoCloseable 인터페이스

- AutoCloseable 인터페이스를 구현한 클래스 만들기

```
public class AutoCloseObj implements AutoCloseable {
 @Override
 public void close( ) throws Exception {
    System.out.println("리소스가 close( ) 되었습니다");
 }
}
```

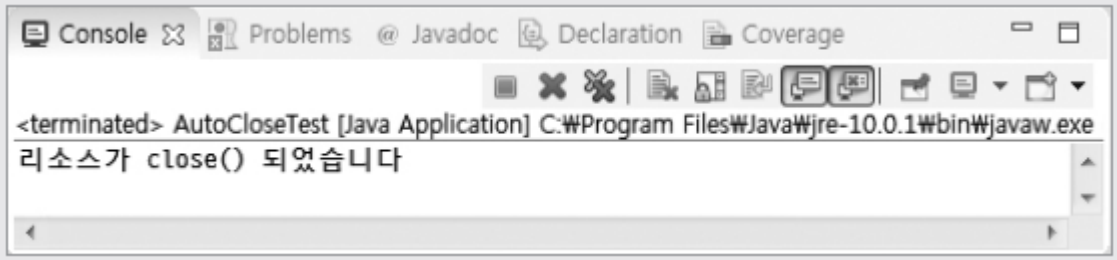
close( ) 메서드 구현

try-with-resources문 사용하기(1)

- 정상적으로 수행 된 경우 : close() 메서드가 호출됨

```
public class AutoCloseTest {
   public static void main(String[ ] args) {
      try(AutoCloseObj obj = new AutoCloseObj( )) {
      } catch(Exception e) {
          System.out.println("예외 부분입니다");
      }
   }
}
```

사용할 리소스 선언

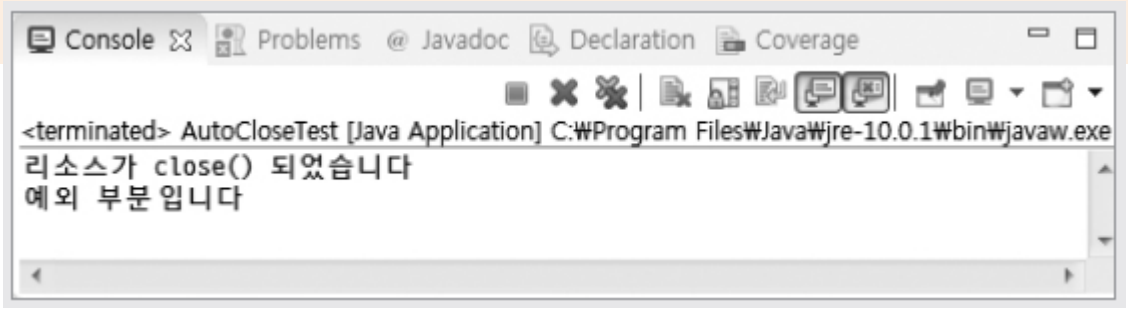


try-with-resources문 사용하기(1)

- 예외가 발생한 경우 : close() 메서드가 호출됨

```
public class AutoCloseTest {
   public static void main(String[ ] args) {
      try (AutoCloseObj obj = new AutoCloseObj( )) {
         throw new Exception( );
      } catch(Exception e) {
         System.out.println("예외 부분입니다");
      }
   }
}
```

강제 예외 발생



향상된 try-with-resources 문

- 자바 9에서 제공되는 구문
- 외부에 선언 된 리소스도 변수만 사용 가능
- 자바 9 이전

```
AutoCloseObj obj = new AutoCloseObj( );
try (AutoCloseObj obj2 = obj)
  throw new Exception( );
}catch(Exception e) {
  System.out.println("예외 부분입니다");
}
```

다른 참조 변수로 다시 선언해야 함

- 자바 9 이후

```
AutoCloseObj obj = new AutoCloseObj( );
try(obj) {
  throw new Exception( );
} catch(Exception e) {
  System.out.println("예외 부분입니다");
}
```

외부에서 선언한 변수를 그대로 쓸 수 있음

예외 처리 미루기

- throws를 사용하여 예외처리 미루기
- 메서드 선언부에 throws를 추가
- 예외가 발생한 메서드에서 예외 처리를 하지 않고,
  이 메서드를 호출한 곳에서 예외 처리를 한다는 의미
- main()에서 throws를 사용하면 가상머신에서 처리됨

예외 처리 미루기 예제

```
public class ThrowsException {
   public Class loadClass(String fileName, String className) throws
      FileNotFoundException, ClassNotFoundException {
      FileInputStream fis = new FileInputStream(fileName);
      Class c = Class.forName(className);
      return c;
   }

   public static void main(String[ ] args) {
      ThrowsException test = new ThrowsException( );
      test.loadClass("a.txt", "java.lang.String");
   }
}
```

두 예외를 메서드가 호출될 때 처리하도록 미룸

FileNotFoundException 발생 가능

ClassNotFoundException 발생 가능

메서드를 호출할 때 예외를 처리함

예외 처리 미루기 예제

- 모든 예외를 한 블록에서 처리하기

```
public static void main(String[ ] args) {
 ThrowsException test = new ThrowException( );
 try {
  test.loadClass("a.txt", "java.lang.String");
 } catch (FileNotFoundException | ClassNotFoundException e) {
  // TODO Auto-generated catch block
  e.printStackTrace( );
 }
}
```

생성됨

여러 예외를 한 문장으로 처리함

예외 처리 미루기 예제

- 각 상황 마다 예외 처리하기

```
public static void main(String[ ] args) {
  ThrowsException test = new ThrowException( );
  try {
    test.loadClass("a.txt", "java.lang.String");
  } catch (FileNotFoundException e) {
    //TODO Auto-generated catch block
    e.printStackTrace( );
  } catch (ClassNotFoundException e) {
    //TODO Auto-generated catch block
    e.printStackTrace( );
  }
}
```

생성됨

각 예외 상황마다
다른 방식으로 처리함

다중 예외 처리 시 주의 사항

- 예외가 다양한 경우,
  가장 최상위 클래스인 Exception 클래스에서 예외를 처리 할 수 있음

```
public static void main(String[ ] args) {
 ThrowsException test = new ThrowsException( );
 try {
    test.loadClass("a.txt", "java.lang.String");
 } catch (FileNotFoundException e) {
    e.printStackTrace( );
 } catch (ClassNotFoundException e) {
    e.printStackTrace( );
 } catch(Exception e) {
    e.printStackTrace( );
 }
}
```

Exception 클래스로
그 외 예외 상황 처리

- 단, Exception **클래스는** 모든 예외 클래스의 최상위 클래스이므로
  가장 마지막 블록에 위치 해야 함

사용자 정의 예외

- JDK 에서 제공되는 예외 클래스 외에 사용자가 필요에 의해 예외 클래스를 정의하여 사용
- 기존 JDK 예외 클래스 중 가장 유사한 클래스에서 상속
- 기본적으로 Exception에서 상속해도 됨

```
public class IDFormatException extends Exception {
 public IDFormatException(String message) {
    super(message);
 }
}
```

생성자의 매개변수로 예외 상황 메시지를 받음

사용자 정의 예외 클래스 예제

- 전달 받은 아이디의 값이 null 이거나 8자 이상 20자 이하가 아닌 경우 예외를 발생시킴

```
public class IDFormatTest {
   private String userID;

   public String getUserID( ) {
      return userID;
   }

   public void setUserID(String userID) throws IDFormatException {
      if(userID == null) {
         throw new IDFormatException("아이디는 null일 수 없습니다");
      }
      else if(userID.length( ) < 8 || userID.length( ) > 20) {
         throw new IDFormatException("아이디는 8자 이상 20자 이하로 쓰세요");
      }
   this.userID = userID;
   }
```

아이디에 대한 제약 조건 구현

IDFormatException 예외를 setUserID( ) 메서드가 호출될 때 처리하도록 미룸

강제로 예외 발생시킴

사용자 정의 예외 클래스 예제

```
    public static void main(String[ ] args) {
        IDFormatTest test = new IDFormatTest( );

        String userID = null;
        try {
            test.setUserID(userID);
        } catch (IDFormatException e) {
            System.out.println(e.getMessage( ));
        }

        userID = "1234567";
        try {
            test.setUserID(userID);
        } catch (IDFormatException e) {
            System.out.println(e.getMessage( ));
        }
    }
}
```

아이디 값이 null인 경우

아이디 값이 8자 이하인 경우

Quiz 비밀번호 예외 클래스

- 사용자 정의 예외 클래스에서 실습한 예외를 응용해 PasswordException을 구현해 보세요.
  - **예외 상황은 비밀번호가 null인 경우**
  - **비밀번호가 문자열로만 이루어진 경우**
  - **비밀번호가 5자 이하인 경우**

- 아래의 힌트를 참고하여 코드를 완성해 예외 처리를 해 보세요~!

```
// 문자열로만 이루어졌는지 확인 : matches() 메서드 사용
String pass = new String("abc");
System.out.println(pass.matches("[a-zA-Z]+"));  // true

String pass2 = new String("abc1");
System.out.println(pass2.matches("[a-zA-Z]+")); // false
```