CSE3040 Java Language

Lecture 13: Exception Handling

Dept. of Computer Engineering,

Sogang University

Errors in Programs

- Compile-time error
  - The compiler produces errors and does not create object files.
  - e.g.) syntax error
- Run-time error
  - The compiler does not complain and creates object files.
  - At run-time, the program cannot continue due to error and thus stops executing.
  - e.g.) trying to call a method of an instance that is null
- Logical error
  - The program runs without error, but the outcome is something that the programmer did not expect
  - e.g.) wrong equations

Run-time Error

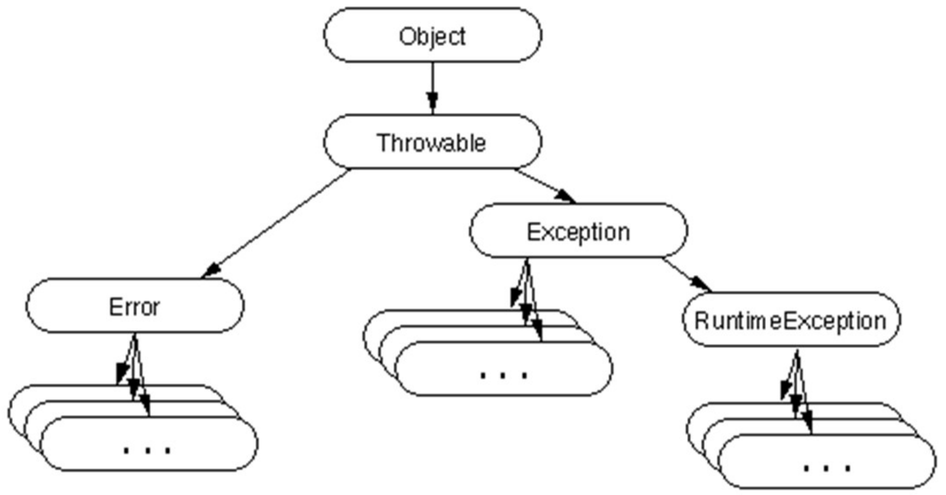
- Run-time errors are categorized into errors and exceptions.

- Errors
  - cannot be recovered and the program must stop.
  - e.g.) out of memory, stack overflow

- Exceptions
  - errors that may be recovered
  - e.g.) arithmetic exception, class cast exception, null pointer exception, index out-of-bounds exception, etc.
  - the programmer may implement handlers that take care of these exceptions when they occur.

Errors and Exceptions

- Java defines errors and exceptions as classes.
  - They are subclasses of class Throwable, which is a subclass of class Object.
  - Class Exception is also divided into two categories
    - Subclasses of RuntimeException and other exception classes
    - RuntimeExceptions: typically caused by programmer mistakes
      - ArrayIndexOutOfBoundsException, NullPointerException, ClassCastException, Arithmetic Exception (e.g. division by zero), etc.
    - Other exceptions: typically caused by environments and user errors.
      - FileNotFoundException, ClassNotFoundException, DataFormatException, etc.



Exception Handling: try-catch

- Errors cannot be handled, but exceptions can be handled.
- By handling exceptions, the program can continue execution without getting crashed.
- In Java, exception handling is done using the try-catch block.

```
try {
    // statements where exceptions can occur.
} catch (Exception1 e1) {
    // statements that will be executed when Exception1 occurs.
} catch (Exception2 e2) {
    // statements that will be executed when Exception2 occurs.
    try {      } catch (Exception3 e3) {     }
    // try-catch blocks can be nested. In that case, the parameters (e2 and e3) must be different.
} catch (ExceptionN eN) {
    // statements that will be executed when ExceptionN occurs.
}
```

Exception Handling: Example

- The following code may cause an exception.
  - (int)(Math.random() * 10) may become zero.
  - If an exception occurs, the program is aborted, and the exception message is printed.

```java
public class Lecture {
    public static void main(String args[]) {
        int number = 100;
        int result = 0;

        for(int i=0; i<10; i++) {
            result = number / (int)(Math.random() * 10);
            System.out.println(result);
        }
    }
}
```

```
33
20
11
25
33
50
Exception in thread "main" java.lang.ArithmeticException: / by zero
     at cse3040/kr.ac.sogang.icsl.Lecture.main(Lecture.java:9)
```

Exception Handling: Example

- We have modified the code to handle the ArithmeticException.
- Now when the divisor becomes 0 (and an ArithmeticException occurs), the catch block is executed.
  - The program continues without being crashed.

```java
public class Lecture {
    public static void main(String args[]) {
        int number = 100;
        int result = 0;

        for(int i=0; i<10; i++) {
            try {
                result = number / (int)(Math.random() * 10);
                System.out.println(result);
            } catch (ArithmeticException e) {
                System.out.println("0");
            }
        }
    }
}
```

Program Flow in a try-catch Block

- If an exception occurs inside a try block:
  - searches for a catch block that catches the corresponding exception.
  - if found, the catch block is executed, and then the flow goes to the statement after the try-catch block.

- If no exception occurs inside a try block:
  - after executing the try block, the flow goes to the statement after the try-catch block.

```
public class Lecture {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(4);
        } catch (Exception e) {
            System.out.println(5);
        }
        System.out.println(6);
    }
}
```

Program Flow in a try-catch Block

```
public class Lecture {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0);
            System.out.println(4);
        } catch (ArithmeticException e) {
            System.out.println(5);
        }
        System.out.println(6);
    }
}
```

The catch Block

- The catch block has a variable of a particular type of exception as its parameter.
- If an exception occurs, the first catch block is checked whether its parameter is a super type of the current exception.
  - If yes, the catch block is executed.
  - If no, the next catch block is checked.
- In the following example, the catch block is executed when the ArithmeticException occurs.
  - Exception is a superclass of ArithmeticException.

```
public class Lecture {
    public static void main(String args[]) {
        try {
            System.out.println(3);
            System.out.println(0/0);    // ArithmeticException!
            System.out.println(4);
        } catch (Exception e) {
            System.out.println(5);
        }
    }
}
```

The catch Block

- In the following example, there are two catch blocks.
- Since the exception is ArithmeticException, the first catch block matches the exception type.
- Thus, the first catch block is executed.
- Then, the flow goes out of the try-catch block.
  - The second catch block is not executed.

```
public class Lecture {
    public static void main(String args[]) {
        try {
            System.out.println(3);
            System.out.println(0/0);    // ArithmeticException!
            System.out.println(4);
        } catch (ArithmeticException e) {
            System.out.println("ArithmeticException occurred!");
        } catch (Exception e) {
            System.out.println("Exception occurred!");
        }
    }
}
```

printStackTrace() and getMessage()

- Often exceptions are caused by bugs, and the programmer wants to know why the exception has occurred.
- printStackTrace and getMessage are two methods that are useful for debugging.
  - printStackTrace: prints the methods that were in the call stack when the exception occurred.
  - getMessage: acquires message that is stored in the exception instance.

```
public class Lecture {
    public static void main(String args[]) {
        try {
            System.out.println(3);
            System.out.println(0/0);    // ArithmeticException!
            System.out.println(4);
        } catch (ArithmeticException ae) {
            ae.printStackTrace();
            System.out.println("exception message: " + ae.getMessage());
        }
    }
}
```

Generating an Exception

- The programmer can deliberately generate an exception using keyword throw.
  - Create an exception instance.
    - Can pass an exception message as an argument to the constructor.
  - Cause (throw) an exception using throw.

```
public class Lecture {
    public static void main(String args[]) {
        try {
            Exception e = new Exception("I created the exception.");  // becomes the exception message.
            throw e;
        } catch (Exception e) {
            System.out.println("exception message: " + e.getMessage());
            e.printStackTrace();
        }
        System.out.println("The program terminated normally.");
    }
}
```

Mandatory and Optional Exception Handling

- If a statement that can cause an exception is not handled, the programmer generates an error.
  - All exceptions except subclasses of RuntimeException
  - These exceptions are called "checked exceptions".

```
public class Lecture {
    public static void main(String args[]) {
        throw new Exception("My exception.");
    }
}  // compile error!
```

- For exceptions that are subclasses of RuntimeException, handling is not mandatory.
  - These exceptions are typically caused by programmer mistakes.
  - Too many statements can throw RuntimeExceptions.
  - These exceptions are called "unchecked exceptions".

```
public class Lecture {
    public static void main(String args[]) {
        throw new ArithmeticException("My exception.");
    }
}  // no error!
```

Methods that throw exceptions

- Instead of using try-catch blocks, we can let a method throw exceptions.
  - It means that these exceptions could occur in this method.
  - If a method throws Exception1, it means that all subclasses of Exception1 can occur in the method.

```
void method() throws Exception1, Exception2, ... , ExceptionN {
    // statements
}
```

- If a method throws an exception, another method that calls this method should handle the exception or throw the exception itself.

```
public class Lecture {
    public static void main(String args[]) {
        method1();
    }
    static void method1() {
        method2();    // Error: must handle Exception or throw Exception.
    }
    static void method2() throws Exception {
        throw new Exception();
    }
}  // compile error!
```

Methods that throw exceptions

- If method1 also throws Exception, then it does not need to handle the Exception.
  - Then, main method that calls method1 must handle the exception or throw the exception.

```
public class Lecture {
    public static void main(String args[]) {
        method1();    // Error: must handle Exception or throw Exception.
    }
    static void method1() throws Exception {
        method2();
    }
    static void method2() throws Exception {
        throw new Exception();
    }
}  // compile error!
```

Methods that throw exceptions

- If main method also throws Exception, then this code will compile without problem
- However, the Exception will cause the program to crash at run time, because it is not handled.

```
public class Lecture {
    public static void main(String args[]) throws Exception {
        method1();
    }
    static void method1() throws Exception {
        method2();
    }
    static void method2() throws Exception {
        throw new Exception();
    }
}  // no error at compile time but will cause program to crash at run time.
```

Methods that throw exceptions

- A proper way of programming is to handle the Exception somewhere in the call stack.
  - case 1: method 2 throws Exception, and method1 handles the exception.
  - case 2: method 2 throws Exception, method 1 throws Exception, and main method handles the exception.

```
public class Lecture {
    public static void main(String args[]) {
        method1();
    }
    static void method1() {
        try {
            method2();
        } catch(Exception e) {
            System.out.println("Exception handled in method1");
            e.printStackTrace();
        }
    }
    static void method2() throws Exception {
        throw new Exception();
    }
}  // OK! method1 handles the Exception
```

Methods that throw exceptions

- In the previous slides (File I/O), we made the main method throw IOException.
- It is because the FileOutputStream constructor throws FileNotFoundException.
  - FileNowFoundException is a subclass of IOException.

```
public FileOutputStream(String name) throws FileNotFoundException
```

- Although the code successfully compiles, this code is not good because it does not actually handle the exception.
  - If the FileNotFoundException occurs, the program will crash.
  - Should use try-catch block to handle the exception.

```
package cse3040;
import java.io.FileOutputStream;
import java.io.IOException;
public class Lecture {
     public static void main(String[] args) throws IOException {
            FileOutputStream output = new FileOutputStream("src/cse3040/out.txt");
            String str = "hello world";
            byte[] bytes = str.getBytes();
            output.write(bytes);
            output.close();
     }
}
```

The finally Block

- In the try-catch block, a finally block can be included at the end.
  - The finally block is executed whether or not an exception is occurred in the try block.

```
try {
    // statements that can cause exceptions.
} catch (Exception1 e1) {
    // statements for handling Exception1
} finally {
    // this block is executed whether or not an exception occurs in the try block.
    // this block must be placed at the end of a try-catch block.
}
```

The finally Block

- Even if there is a return statement in the try block, still the finally block is executed before the method is returned!

```
public class Lecture {
    public static void main(String args[]) {
        Lecture.method1();
        System.out.println("returned to main method after calling method1.");
    }

    static void method1() {
        try {
            System.out.println("the try block of method 1 is being executed.");
            return;
        } catch(Exception e) {
            e.printStackTrace();
        } finally {
            System.out.println("the finally block of method 1 is being executed.");
        }
    }
}
```

서강대학교 SOGANG UNIVERSITY

The finally Block

- When reading from a file, exceptions must be handled if the method does not throw the exceptions.

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

public class Lecture {
    public static void main(String[] args) {
        byte[] b = new byte[1024];
        FileInputStream input = null;
        try {
            input = new FileInputStream("src/kr/ac/sogang/icsl/aaa.txt");
            input.read(b);
            System.out.println(new String(b));
        } catch(IOException e) {
           e.printStackTrace();
        } finally {
            try {
                input.close();
            } catch(Exception e) {
                e.printStackTrace();
            }
        }
        System.out.println("The program exited normally.");
    }
}
```

Programming Lab #13

서강대학교
SOGANG UNIVERSITY

14-01. Defining and Using a Generic Class

- Modify the following code to use a generic class Box instead of BoxA and BoxB.

```
class A { public String toString() { return "Class A Object"; }}
class B { public String toString() { return "Class B Object"; }}
class C { public String toString() { return "Class C Object"; }}

class BoxA {
     A item;
     void setItem(A item) { this.item = item; }
     A getItem() { return item; }
}

class BoxB {
     B item;
     void setItem(B item) { this.item = item; }
     B getItem() { return item; }
}

class BoxC {
     C item;
     void setItem(C item) { this.item = item; }
     C getItem() { return item; }
}
```

14-01. Defining and Using a Generic Class

```
public class Ex14_01 {
    public static void main(String[] args) {
        BoxA boxa = new BoxA();
        boxa.setItem(new A());
        BoxB boxb = new BoxB();
        boxb.setItem(new B());
        BoxC boxc = new BoxC();
        boxc.setItem(new C());

        System.out.println(boxa.getItem());
        System.out.println(boxb.getItem());
        System.out.println(boxc.getItem());
    }
}
```

14-02. try-catch-finally Statement

- Try running the following code and understand the result.
- Try removing the statement that produces the exception and run the code.

```
public class Ex13_02 {
    public static void main(String args[]) {
        System.out.println(1);
        System.out.println(2);
        try {
            System.out.println(3);
            System.out.println(0/0);
            System.out.println(4);
        } catch(ArithmeticException e) {
            System.out.println(5);
        } catch(Exception e) {
            System.out.println(6);
        } finally {
            System.out.println(7);
        }
        System.out.println(8);
    }
}
```

13-03. Throwing Exceptions

- The following code produces a compile error. Fix the error by:
  - Handling the exception in method2.
  - Handling the exception in method1.
  - Handling the exception in the main method.
  - Not handling the exception anywhere.
- Try changing Exception to RuntimeException.

```
public class Ex13_03 {
    public static void main(String args[]) {
        method1();
    }
    static void method1() {
        method2();
    }
    static void method2() {
        throw new Exception();
    }
}
```

서강대학교 SOGANG UNIVERSITY

13-04. Handling File I/O Errors

- The following code produces a run-time error if the file does not exist. Handle the exception so that the program does not crash but terminates normally when the input file is not found.

```
public class Ex13_04 {
    public static void main(String[] args) {
        BufferedReader br = new BufferedReader(new FileReader("src/cse3040ex1304/myFile1.txt"));
        while(true) {
            String line = br.readLine();
            if(line == null) break;
            System.out.println(line);
        }
        br.close();
    }
}
```

End of Class

Instructor office: AS818A

Email: jso1@sogang.ac.kr