Java Programming

Zheng-Liang Lu

Department of Computer Science & Information Engineering
National Taiwan University

Java 415
Summer 2024

```java
class Lecture8 {

            "Exceptions and Exception Handling"

}

// Keywords:
try, catch, finally, throw, throws, assert
```

Introduction

- An exception is to interrupt "normal" program flows.[1]
  - For example, opening a non-existing file results in **FileNotFoundException**.
- When the callee throws an exception object, this object should be well-handled by the caller, by providing proper exception handlers.
- In other words, a specific exception handler catches the associated exception.

[1]Note that an exception should be a force majeure event.

The Handling Block: try-catch-finally

- Now we proceed to introduce the three components of exception handlers: the try, catch, and finally blocks.
- First, wrap the normal operations which may throw exceptions in the try block.
- We then write down the handlers for specific exceptions.[2]
  - You may consider a multi-catch (using | to separate them).[3]
  - Usually, we put the super-type **Exception** in the last catch clause to catch the exceptional exceptions.
- Java provides the finally block, which is always executed when the try block exits.
  - This block is mainly used for cleanup, say closing a file.

[2]Try to handle each exception but not once at all.

[3]The grouped exceptions in the same catch clause should be siblings.

```java
import java.util.Scanner;
import java.util.InputMismatchException;

public class ExceptionDemo {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);

        try {
            System.out.println("Enter an integer?");
            int x = input.nextInt();
        } catch (InputMismatchException e) {
            System.out.println("Not an integer.");
        } catch (Exception e) {
            System.out.println("Unknown exception.");
        } finally {
            input.close();
            System.out.println("Cleanup is done.");
        }

        System.out.println("End of program.");
    }

}
```

Exception Hierarchy[4]

- The topmost class of the exception hierarchy is **Throwable**.
- All **Throwable** subclasses are categorized into two groups: unchecked exceptions and checked exceptions.
- Checked exceptions must be checked at compile time.
  - For example, **IOException** and **Exception**.
- Unchecked exceptions are not forced by the compiler to either handle or specify the exception.
  - For example, **RuntimeException**.

[4] See Diagram of Exception Hierarchy.

Throwing Exceptions

- As a library maker, we disallow some user's behaviors.
- Java provides the throwing mechanism by using throw (issuing) and throws (translation).

```java
public class Circle {

    private double radius;

    public Circle(double r) throws Exception {

        if (r <= 0) throw new Exception("Invalid radius.");
        radius = r;

    }

}
```

Customized Exceptions

- Use class inheritance to create our own exceptions.

```java
public class InvalidRadiusException extends Exception {

    public InvalidRadiusException(double r) {

        super(String.valueOf(r));

    }

}
```

```java
public class Circle {

    private double radius;

    public Circle(double r) throws InvalidRadiusException {

        if (r <= 0) throw new InvalidRadiusException(r);
        radius = r;

    }

}
```

```java
public class NewExceptionDemo {

    public static void main(String[] args) {

        try {
            new Circle(-10);
        } catch (InvalidRadiusException e) {
            System.out.println(e); // Check the result!
        }

    }

}
```

Assertion

- An assertion is a statement that enables you to test your assumption about the program, as an internal check.
- Before running the program, add "-ea" to the VM arguments so that these assertion statements can be tested.

```java
public class AssertionDemo {

    public static void main(String[] args) {

        int x = 1;
        assert("x is not equal to 2.", x == 2);
        // AssertionError occurs!!
        System.out.println("End of program.");

    }

}
```

Unit Test: JUnit

- Writing test codes is to automate the testing routine for future changes.
  - What works in the past should work after modification.[5]
- However, we should avoid writing test codes together with the normal codes!
- In practice, you may use JUnit[6] to write test cases for your project.

[5] See also Test-Driven Development (TDD).

[6] See https://junit.org/.

Fin.