CSE201: Monsoon 2017
Advanced Programming

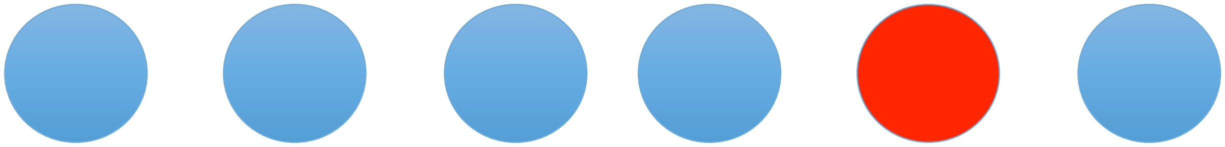
Lecture 17: Introduction to Exceptions

Vivek Kumar
Computer Science and Engineering
IIIT Delhi
vivekk@iiitd.ac.in

Last Lecture

- Defensive Programming
  - Collection of techniques to reduce the risk of failure at run time
- Rules
  - Never assume anything
    - Take care of invalid inputs
      - No Garbage in => Garbage out
  - Follow proper coding standards
    - Create and follow programming standards
    - Don’t use magic numbers
    - Use proper indentations
  - Keep your code as simple as possible
    - Functions should be seen as contract. Given input they should a specific task
    - Code refactoring
    - Code reuse

Today's Lecture: Exceptions



Types of Programming Errors

- Syntax errors
  - Compile time errors
  - Easiest to fix
- Logical errors
  - Program runs without crashing but gives incorrect result
  - Most difficult to fix
- Runtime errors
  - Occur while the program is running if the environment detects an operation that is impossible to carry out
  - Could be fixed easily with defensive programming
    - **Exception handling!**

Exception Handling Syntax

- Process for handling exceptions
  - `try` some code, catch exception thrown by tried code, finally, “clean up” if necessary
  - `try`, `catch`, and `finally` are reserved words
- `try` denotes code that may throw an exception
  - place questionable code within a `try` block
  - a `try` block must be immediately followed by a `catch` block unlike an if w/o else
  - thus, `try-catch` blocks always occurs as pairs
- `catch` exception thrown in `try` block and write special code to handle it
  - catch blocks distinguished by type of exception
  - can have several ***catch blocks***, each specifying a particular type of exception
  - Once an exception is handled, execution continues after the catch block
- `finally` (optional)
  - special block of code that is executed whether or not an exception is thrown
  - follows *catch block*

Trace a try/catch Program Execution (1/3)

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose no exceptions in the statements

Trace a `try/catch` Program Execution (2/3)

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed

Trace a try/catch Program Execution (3/3)

```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Next statement in the method is executed

Trace a try/catch Program Execution (1/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

Suppose an exception of type Exception1 is thrown in statement2

Trace a try/catch Program Execution (2/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The exception is handled.

Trace a try/catch Program Execution (3/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The final block is always executed.

Trace a try/catch Program Execution (4/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
finally {
  finalStatements;
}

Next statement;
```

The next statement in the method is now executed.

Is this Defensive Programming ?

```
import java.util.*;
public class Main {

    public static void main(String[] args) {


            System.out.println(“Enter Integer Input”);

                Scanner sc = new Scanner(System.in);
                int num = sc.nextInt();







        }
    }
}
```

- Is program correct?
  - Yes
    - But, only if the user is paying attention
      - Invalid input ?
      - String as input?

Exception Handling using try/catch

```
import java.util.*;
public class Main {

    public static void main(String[] args) {
        boolean done = false;
        while(!done) {
            System.out.println(“Enter Integer Input”);
            try {
                Scanner sc = new Scanner(System.in);
                int num = sc.nextInt(); //exception point
                done = true;
            }
            catch(InputMismatchException inp) {
                System.out.println(“Wrong input:”);
                System.out.println(“Try again”);
            }
            finally {
                System.out.println(“Always execute”);
            }
        }
    }
}
```

- This is a foolproof program now!
- Exception handling using `try/catch` block of statements
  - Defensive programming
- `InputMismatchException` is a type of exception provided by the `Scanner` class in Java

Multiple catch Blocks

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        String[] s = {“a”, “23”, null, “4”, “P”};
        int sum = 0;
        for(int i=0; i<10; i++) {
            try {
                sum += (s[i].length() > 0) ?
                           Integer.parseInt(s[i]) : 0;
            }
            catch(NumberFormatException e) {
                System.out.println(“Not an Integer”);
            }
            catch(NullPointerException e) {
                System.out.println(“NULL value found”);
            }
            catch(ArrayIndexOutOfBoundsException e) {
                System.out.println(“Index not in range”);
            }
        }
    }
}
```

- There could be multiple catch for a single try block
- They are designed to catch different types of exceptions that could be raised from a single try block
- **How the exceptions are generated here?**
  - i=0 will raise NumberFromatException
  - i=2 will raise NullPointerException
  - i=4 will raise NumberFormatException
  - i>4 will raise ArrayIndexOutOfBounds exception

Question

```
public class Main {
    public static void main(String[] args) {
        String s = null;
        try {
            int length = s.length();
        }

        System.out.println(“Just before catch block”);

        catch(NullPointerException e) {
            System.out.println(“String was null”);
        }
    }
}
```

- What is the output of the following program?
- Answer
    - ○ Compilation error!
    - ○ **No statement is allowed between a pair of try and catch**
    - ○ `error: 'catch' without 'try'`

Nested try/catch Blocks

```
public class Andy {
    .....

    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);

            try {
                _water = johny.getADrink();
                int volume = _water.getVolume();
            }
            catch(NullPointerException e) {
                this.fire(johny);
            }
        }
    }
}
```

- `try/catch` block could be nested!
    - ○ If Andy's call to `getADrink` from Wendy returns null, he can ask Johny to `getADrink`

Methods Can throw Exception

```
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NullPointerException(“NO Water”);
            // Although the below throw is correct
            // but its not of any help!!
            // throw NullPointerException(“Error”);
        }
    }
}
```

- If you wish to throw an exception in your code you use the **throw** keyword
- Most common would be for an unmet precondition
- When the program detects an error, the program can create an instance of an appropriate exception type and throw it:

```
throw new TheException(“Message”);
```

- In the above constructor call for the exception, the message is optional but it’s always good to pass some meaningful message

Re-throwing Exception

```
public class Andy {
    .....
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NullPointerException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);
            System.out.println(“Wendy is fired!”);
            throw new NullPointerException(“NO Water”);
        }
    }
}
```

- The caught exceptions can be re-thrown using **`throw`** keyword
- Re-thrown exception must be handled some where in the program, otherwise program will terminate abruptly

Trace a try/catch Program Execution (1/4)

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
catch(Exception2 ex) {
   handling ex;
   throw ex;
}
finally {
   finalStatements;
}

Next statement;
```

statement2 throws an exception of type Exception2.

Trace a try/catch Program Execution (2/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception

Trace a try/catch Program Execution (3/4)

```
try {
   statement1;
   statement2;
   statement3;
}
catch(Exception1 ex) {
   handling ex;
}
catch(Exception2 ex) {
   handling ex;
   throw ex;
}
finally {
   finalStatements;
}

Next statement;
```

Execute the final block

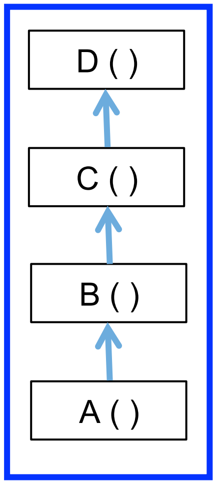
Trace a try/catch Program Execution (4/4)

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller

How Exceptions are Handled by JVM



- Any method invocation is represented as a “**stack frame**” on the Java “**stack**”
  - **Callee-Caller** relationship
    - If method A calls method B then A is **caller** and B is **callee**
  - Each frame stores local variables, input parameters, return values and intermediate calculations
    - In addition, each frame also stores an “**exception table**”
    - This exception table stores information on each `try/catch/finally` block, i.e. the instruction offset where the `catch/finally` blocks are defined
  - When an exception is thrown, JVM does the following:
    1. Look for exception handler in current stack frame (method)
    2. If not found, then terminate the execution of current method and go to the callee method and repeat step 1 by looking into callee’s exception table
    3. If no matching handler is found in any stack frame, then JVM finally terminates by throwing the stack trace (`printStackTrace` method)

Next Lecture

- Exceptions (continued)
- Assertions