CSE201: Monsoon 2017
Advanced Programming

Lecture 18: Exceptions (contd.) & Assertions

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

Last Lecture

- Exception handling
  - To catch runtime errors
  - **try / catch / finally** block to exception handling
  - **try/catch** blocks could be nested
  - Single **try** could have multiple **catch** blocks
  - Methods can **throw** exceptions

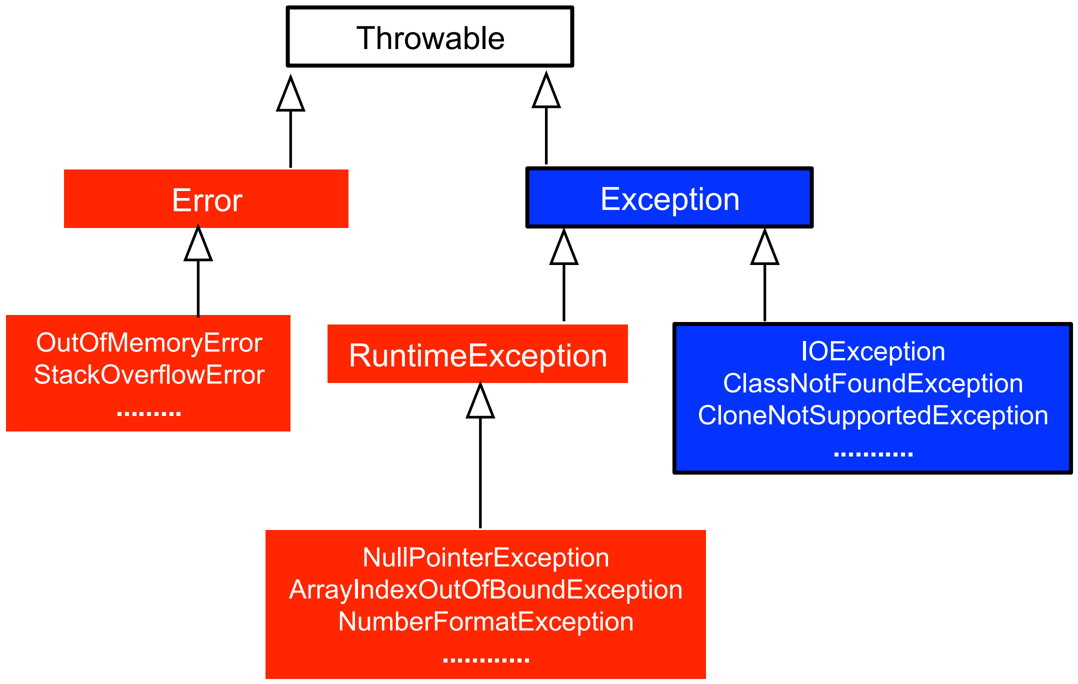
```
public class Andy {
    public void getWater() {
        try {
            _water = _wendy.getADrink();
            int volume = _water.getVolume();
        }
        catch(NullPointerException e) {
            this.fire(_wendy);

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

```
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");
        }
    }
}
```

Today’s Lecture

- Exceptions (continued from last lecture)
- Assertions

Exception Hierarchy



- `Exceptions` are classes that extends `Throwable`
- Come in two types
  - **Checked exceptions**
    - Those that must be handled somehow (we will see soon)
      - E.g., `IOException` – file reading issue
  - **Unchecked exceptions**
    - Those that do not
      - E.g., `RuntimeExceptions` that is caused due to programming errors
      - You should not attempt to handle exceptions from subclass of `Error`
        - Rarely occurring exceptions that even if you try to handle, there is little you can do beyond notifying the user and trying to terminate the program gracefully

Handling Checked Exception (1/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName)  {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- If we have code that tries to build a FileReader we must deal with the possibility of the exception
- The code contains a syntax error. "`unreported exception java.io.FileNotFoundException`
  - must be caught or declared to be thrown

Handling Checked Exception (2/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName)  {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- Here, there are 4 statements that can generate checked exceptions:
  - The FileReader constructor
  - the ready method
  - the read method
  - the close method
- To deal with the exceptions we can either state this method **”throws”** an Exception of the proper type or handle the exception within the method itself

Handling Checked Exception (3/3)

```
import java.io.FileReader;

public class Tester {
    public int countChars(String fileName)  throws
FileNotFoundException, IOException {
        FileReader r = new FileReader(fileName);
        int total = 0;
        while( r.ready() ) {
            r.read();
            total++;
        }
        r.close();
        return total;
    }
}
```

- It may be that we don't know how to deal with an error within the method that can generate it
- In this case we will pass the buck to the method that called us
- The keyword **`throws`** is used to indicate a method has the possibility of generating an exception of the stated type
- Now any method calling ours must also throw an exception or handle it

Question

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

        catch (Exception e) {
            System.out.println("Catch block -1");
        }
        catch (NullPointerException e) {
            System.out.println("Catch block -2");
        }
    }
}
```

- What is the output of the following program?
- Answer
  - Compilation error!
  - **Unreachable catch block**
  - `error: exception NullPointerException has already been caught`

Some Important Methods in `Throwable`

`String toString()`	Returns a short description of the exception
`String getMessage()`	Returns the detail description of the exception
`void printStackTrace()`	Prints the stacktrace information on the console

```
1. public class Andy {

2.     public void drinkWater() {
3.         getWater();
4.     }
5.     public void getWater() {
6.         try {
7.             _water = _wendy.getADrink();//null
8.             int volume = _water.getVolume();
9.         }
10.         catch(NullPointerException e) {
11.             e.printStackTrace();
12.         }
13.     }
14. }
```

- Output:

```
java.lang.NullPointerException
    at Andy.getWater(Andy.java:8)
    at Andy.drinkWater(Andy.java:3)
    ......
```

Overriding Methods Having `throws` (1/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
                  throws CloneNotSupportedException {

        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
    {

        System.out.println("Cloning not allowed");
    }
}
```

- If a method in parent class throws an exception (either checked or unchecked), then overridden implementation of that method in child class is not required to throw that exception
  - Although throwing that **same** exception in overridden method won't hurt

Overriding Methods Having throws (2/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {

        System.out.println(“Clone created”);
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
                 throws CloneNotSupportedException {

        System.out.println(“Cloning not allowed”);
    }
}
```

- However, the reverse may/may not work
- Case-1: Overridden method throws **checked exception** but not the actual method in parent class
    - Compilation error

Overriding Methods Having `throws` (3/3)

```
import java.lang.CloneNotSupportedException;

public class Cloning {

    public void createClone()
    {

        System.out.println("Clone created");
    }
}

public class Human extends Cloning {

    @Override
    public void createClone()
                    throws RuntimeException {

        System.out.println("Cloning not allowed");
    }
}
```

- However, the reverse may/may not work
- Case-2: Overridden method throws **unchecked exception** but not the actual method in parent class
  - This works fine

Defining Your Own Exception (1/4)

```
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NoWaterException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
}
```

- You can define and throw your own specialized exceptions
    - `throw new NoWaterException(…);`
- Useful for responding to special cases, not covered by pre-defined exceptions
- The class `Exception` has a method `getMessage()`.The String passed to `super` is printed to the output window for debugging when `getMessage()` is called by the user

Defining Your Own Exception (2/4)

```
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() {
        try {
            getWater();
        }
        catch(NoWaterException e) {
            System.out.println(e.getMessage());
        }
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException(“NO Water”);
        }
    }
}
```

- Every method that throws `Exceptions` that are not subclasses of `RuntimeException` must declare what exceptions it throws in method declaration
- `getWater()` is throwing the exception, hence it must declare that using the “`throws`” on method declaration

Defining Your Own Exception (3/4)

```
public class NoWaterException extends Exception {
    public NoWaterException(String message) {
        super(message);
    }
}
public class Andy {
    public void drinkWater() throws NoWaterException {
        getWater();
    }
    public void getWater() throws NoWaterException {
        _water = _wendy.getADrink();
        if(_water == null) {
            this.fire(_wendy);
            throw new NoWaterException("NO Water");
        }
    }
    public static void main(String[] args) {
        Andy obj = new Andy();
        obj.drinkWater();
    }
}
```

- Any method that directly or indirectly calls `getWater()` must declare that it can generate `NoWaterException` using `throws` keyword
    - Not doing this generate compilation error
    - `error: unreported exception NoWaterException; must be caught or declared to be thrown`

Defining Your Own Exception (4/4)

```
1.public class NoWaterException extends Exception {
2.    public NoWaterException(String message) {
3.        super(message);
4.    }
5.}
6.public class Andy {
7.    public void drinkWater() throws NoWaterException {
8.        getWater();
9.    }
10.    public void getWater() throws NoWaterException {
11.        _water = _wendy.getADrink();
12.        if(_water == null) {
13.            this.fire(_wendy);
14.            throw new NoWaterException(“NO Water”);
15.        }
16.    }
17.    public static void main(String[] args)
18.                          throws NoWaterException {
19.        Andy obj = new Andy();
20.        obj.drinkWater();
21.    }
22.}
```

- This works fine, although we are not catching the `NoWaterException` anywhere that is again not a defensive programming!
  - Running this program with `_water = null`

```
Exception in thread "main”
NoWaterException: NO Water

at Andy.getWater(Andy.java:14)

at Andy.drinkWater(Andy.java:8)

at Andy.main(Andy.java:20)
```

Pros and Cons of Exception

- Pros
  - Cleaner code: rather than returning a boolean up chain of calls to check for exceptional cases, throw an exception!
  - Use return value for meaningful data, not error checking
  - Factor out error-checking code into one class, so it can be reused
- Cons
  - Throwing exceptions requires extra computation
  - Can become messy if not used economically
  - Can accidentally cover up serious exceptions, such as `NullPointerException` by catching them

Let’s change gears…

Assertions

- **assertion**: A statement that is either true or false

  Examples:
  - Java was created in 1995.
  - The sky is purple.
  - 23 is a prime number.
  - 10 is greater than 20.
  - x divided by 2 equals 7. *(depends on the value of x)*

- An assertion might be false ("The sky is purple" above), but it is still an assertion because it is a true/false statement

Declaring Assertions

An *assertion* is declared using the new Java keyword assert as follows:

assert *assertion*; or
assert *assertion : detailMessage*;

where **assertion** is a Boolean expression and ***detailMessage*** is a primitive-type or an Object value

Executing Assertion (1/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

- When an assertion statement is executed, Java evaluates the assertion. If it is false, an `AssertionError` will be thrown

Executing Assertion (2/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    assert i == 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

- By default, the assertions are disabled at runtime as they are costly
  - Constant check of the condition inside assert statement
- To enable use the following command line switch

  `java –ea AssertionDemo`

  OR

  `java –enableassertions AssertionDemo`

Executing Assertion (3/3)

```
public class AssertionDemo {
  public static void main(String[] args) {
    int i; int sum = 0;
    for (i = 0; i < 10; i++) {
      sum += i;
    }
    // deliberately changed to generate assertion failure
    assert i != 10;
    assert sum > 10 && sum < 5 * 10 : "sum is " + sum;
  }
}
```

- Let’s try to generate the assertion failure in this program
  - Change “==“ to “!=“
  - Output:

```
Exception in thread "main" java.lang.AssertionError
at AssertionDemo.main(AssertionDemo.java:7)
```

- `AssertionError` extends `Error` and you cannot write a `try/catch` block to catch this. The program will definitely terminate with the complete stack dump

Assertions or Exception Handling? (1/2)

- Assertion should not be used to replace exception handling
  - Exception handling deals with unusual circumstances whereas assertions are to assure the correctness of the program
  - Exception handling addresses robustness and assertion addresses correctness
- Similar to exceptions, assertions are also checked at runtime but unlike exceptions it can be turned on or off (for entire execution)
- Use assertions to reaffirm assumptions to assure correctness of the program

Assertions or Exception Handling? (2/2)

```
switch (month) {
  case 1: ... ; break;
  case 2: ... ; break;
  ...
  case 12: ... ; break;
  default: assert false : "Invalid month: " + month;
}
```

- Another good use of assertions is to place assertions in a switch statement without a default case

Next Lecture

- Java File IO