Ollscoil na Gaillimhe
University of Galway

CT326 Programming III

LECTURE 5

THROWING & HANDLING EXCEPTIONS

DR ADRIAN CLEAR
SCHOOL OF COMPUTER SCIENCE

Ollscoil na Gaillimhe
University of Galway

Objectives for today

- Understand the uses of exceptions in Java
- See how to create your own exceptions
- Demonstrate how to throw and catch exceptions

Uses of exception handling

- When method cannot complete its task
- Process exceptions from program components
- Uniformity in documenting, detecting, and recovering from errors
  - Useful for understanding error-processing code in large projects

Exception Handling Should Be Used!

Other Error-Handling Techniques

- Using no error-handling
  - Not for mission critical applications
- Exit application on error
  - Program must return resources

Basics of Java Exception Handling

- A method detects an error and throws an exception
  - Exception handler processes the error
    - The error is considered caught and handled in this model
- Code that could generate errors put in **`try`** blocks
  - Code for error handling enclosed in a **`catch`** block
  - The **`finally`** always executes with or without an error
- Keyword **`throws`** specifies exceptions a method might throw if a problem occurs
- Termination model of exception handling
  - The block in which the exception occurs expires

`try` Blocks

- The **`try`** block structure

```
try {
  statements that may throw an exception
}
catch ( ExceptionType exceptionReference ) {
  statements to process an exception
}
```

- A **`try`** is followed by any number of **`catch`** blocks

Throwing an Exception

- The **`throw`** statement
  - Indicates an exception has occurred
  - Operand is any class derived from **`Throwable`**
- Subclasses of **`Throwable`**
  - Class **`Exception`**
    - Problems that should be caught
  - Class **`Error`**
    - Serious exception should not be caught
- Control moves from **`try`** block to catch **`block`**

Catching an Exception

- Handler catches exception
  - Executes code in **`catch`** block
  - Should only catch **`Exception`**s
- Program terminates if no appropriate handler
- Single **`catch`** can handle multiple exceptions
- Many ways to write exception handlers
- Rethrow exception if **`catch`** cannot handle it

`throws` Clause

- Lists the exceptions thrown by a method

```
int functionName ( paramterList )
  throws ExceptionType1, ExceptionType2,…
{
  // method body
}
```

- **`RuntimeException`**s occur during execution
  - **`ArrayIndexOutOfBoundsException`**
  - **`NullPointerException`**
- Declare exceptions a method throws

Checked and unchecked exceptions

- **Unchecked exceptions** (or runtime exceptions) represent unrecoverable errors that occur during the execution of a program
  - subclasses of `RuntimeException` (e.g. `NullPointerException`)
  - Not necessary to add a throws declaration
  - Not necessary to handle
- **Checked exceptions** are caught at compile time
  - All exceptions other than subclasses of `RuntimeException` (e.g. `FileNotFoundException`)
  - Predictable and recoverable
  - must be handled in the method body, either with a try/catch statement or by re-throwing.
  - Checked exceptions need throws declaration.

Finalizers, and Exception Handling

- Throw exception if constructor causes error
- **`Finalize`** called when object garbage collected
- Inheritance of exception classes
  - Allows polymorphic processing of related exceptions

`finally` Block

- Resource leak
  - Caused when resources are not released by a program
- The **`finally`** block
  - Appears after **`catch`** blocks
  - Always executes
  - Use to release resources

```
1    // Fig. 14.9: UsingExceptions.java
2    // Demonstration of the try-catch-finally
3    // exception handling mechanism.
4    public class UsingExceptions {
5
6       // execute application
7       public static void main( String args[] )
8       {
9          // call method throwException
10         try {
11            throwException();
12         }
13
14         // catch Exceptions thrown by method throwException
15         catch ( Exception exception )
16         {
17            System.err.println( "Exception handled in main" );
18         }
19
20         doesNotThrowException();
21      }
22
23      // demonstrate try/catch/finally
24      public static void throwException() throws Exception
25      {
26         // throw an exception and immediately catch it
27         try {
28            System.out.println( "Method throwException" );
29            throw new Exception();  // generate exception
30         }
31
```

Method **main** immediately enters **try** block

Calls method **throwException**

Handle exception thrown by **throwException**

Call method **doesNotThrow-Exception**

Method throws new **Exception**

```
32          // catch exception thrown in try block
33          catch ( Exception exception )
34          {
35             System.err.println(
36                "Exception handled in method throwException" );
37             throw exception;  // rethrow for further processing
38
39             // any code here would not be reached
40          }
41
42          // this block executes regardless of what occurs in
43          // try/catch
44          finally {
45             System.err.println(
46                "Finally executed in throwException" );
47          }
48
49          // any code here would not be reached
50       }
51
52       // demonstrate finally when no exception occurs
53       public static void doesNotThrowException()
54       {
55          // try block does not throw an exception
56          try {
57             System.out.println( "Method doesNotThrowException" );
58          }
59
60          // catch does not execute, because no exception thrown
61          catch( Exception exception )
62          {
63             System.err.println( exception.toString() );
64          }
65
```

Catch **`Exception`**

Rethrow **`Exception`**

The **`finally`** block executes, even though **`Exception`** thrown

Skip **`catch`** block since no **`Exception`** thrown

```
66          // this block executes regardless of what occurs in
67          // try/catch
68          finally {
69             System.err.println(
70                "Finally executed in doesNotThrowException" );
71          }
72
73          System.out.println(
74             "End of method doesNotThrowException" );
75       }
76
77    }  // end class UsingExceptions
```

The **`finally`** block always executes

```
Method throwException
Exception handled in method throwException
Finally executed in throwException
Exception handled in main
Method doesNotThrowException
Finally executed in doesNotThrowException
End of method doesNotThrowException !
```

14.14 Using `printStackTrace` and `getMessage`

- Method **`printStackTrace`**
  - Prints the method call stack
- **`Throwable`** class
  - Method **`getMessage`** retrieves **`informationString`**

```
1    // Fig. 14.11: UsingExceptions.java
2    // Demonstrating the getMessage and printStackTrace
3    // methods inherited into all exception classes.
4    public class UsingExceptions {
5
6       // execute application
7       public static void main( String args[] )
8       {
9          // call method1
10         try {
11            method1();
12         }
13
14         // catch Exceptions thrown from method1
15         catch ( Exception exception ) {
16            System.err.println( exception.getMessage() + "\n" );
17            exception.printStackTrace();
18         }
19      }
20
21      // call method2; throw exceptions back to main
22      public static void method1() throws Exception
23      {
24         method2();
25      }
26
27      // call method3; throw exceptions back to method1
28      public static void method2() throws Exception
29      {
30         method3();
31      }
32
```

Error information generated by **`getMessage`** and **`printStackTrace`**

```
33      // throw Exception back to method2
34      public static void method3() throws Exception
35      {
36         throw new Exception( "Exception thrown in method3" );
37      }
38
39   }  // end class Using Exceptions
```

Throw an **`Exception`** that propagates back to **`main`**

```
Exception thrown in method3

java.lang.Exception: Exception thrown in method3
        at UsingExceptions.method3(UsingExceptions.java:36)
        at UsingExceptions.method2(UsingExceptions.java:30)
        at UsingExceptions.method1(UsingExceptions.java:24)
        at UsingExceptions.main(UsingExceptions.java:11))
```

Ollscoil na Gaillimhe
University of Galway

In-class demonstration

Account

- Let's refactor our code so that the setting of the balance is done by a separate private method, `setBalance`
- `setBalance` should throw a `LessThanZeroException` if the balance is negative
- Refactor the `makeWithdrawal` method so that it uses `setBalance`

```
16
17  public void makeWithdrawal(float anAmount) throws InsufficientFundsException {
18      try {
19          setBalance(balance - anAmount);
20      } catch (LessThanZeroException ex){
21          throw new InsufficientFundsException("Cannot withdraw more than the balance.");
22      }
23  }
24
25  private void setBalance(float balance) throws LessThanZeroException {
26      if (balance >= 0)
27          this.balance = balance;
28      else
29          throw new LessThanZeroException("Cannot set a negative balance.");
30  }
31
```

```
1  package exceptions;
2
3  public class LessThanZeroException extends Exception {
4
5      public LessThanZeroException(String message) {
6          super(message);
7      }
8
9  }
```

Next time…

- Handling Strings in Java