Exception Handling

Advanced Programming

ICOM 4015

Lecture 15

Reading: Java Concepts Chapter 15

Chapter Goals

- To learn how to throw exceptions
- To be able to design your own exception classes
- To understand the difference between checked and unchecked exceptions
- To learn how to catch exceptions
- To know when and where to catch an exception

Error Handling

- **Traditional approach: Method returns error code**
- **Problem: Forget to check for error code**
  - Failure notification may go undetected
- **Problem: Calling method may not be able to do anything about failure**
  - Program must fail too and let its caller worry about it
  - Many method calls would need to be checked

Continued…

Error Handling

- **Instead of programming for success**

```
x.doSomething()
```

  - you would always be programming for failure:

```
if (!x.doSomething()) return false;
```

Throwing Exceptions

- **Exceptions:**
  - Can't be overlooked
  - Sent directly to an exception handler–not just caller of failed method
- **Throw an exception object to signal an exceptional condition**
- **Example: `IllegalArgumentException`:**

```
illegal parameter value
IllegalArgumentException exception
   = new IllegalArgumentException("Amount exceeds balance");
throw exception;
```


Continued…

Throwing Exceptions

- **No need to store exception object in a variable:**

```
throw new IllegalArgumentException("Amount exceeds balance");
```

- **When an exception is thrown, method terminates immediately**
  - Execution continues with an exception handler

Example

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
               = new IllegalArgumentException("Amount
                  exceeds balance");
      throw exception;
      }
      balance = balance - amount;
   }
   . . .
}
```

Hierarchy of Exception Classes

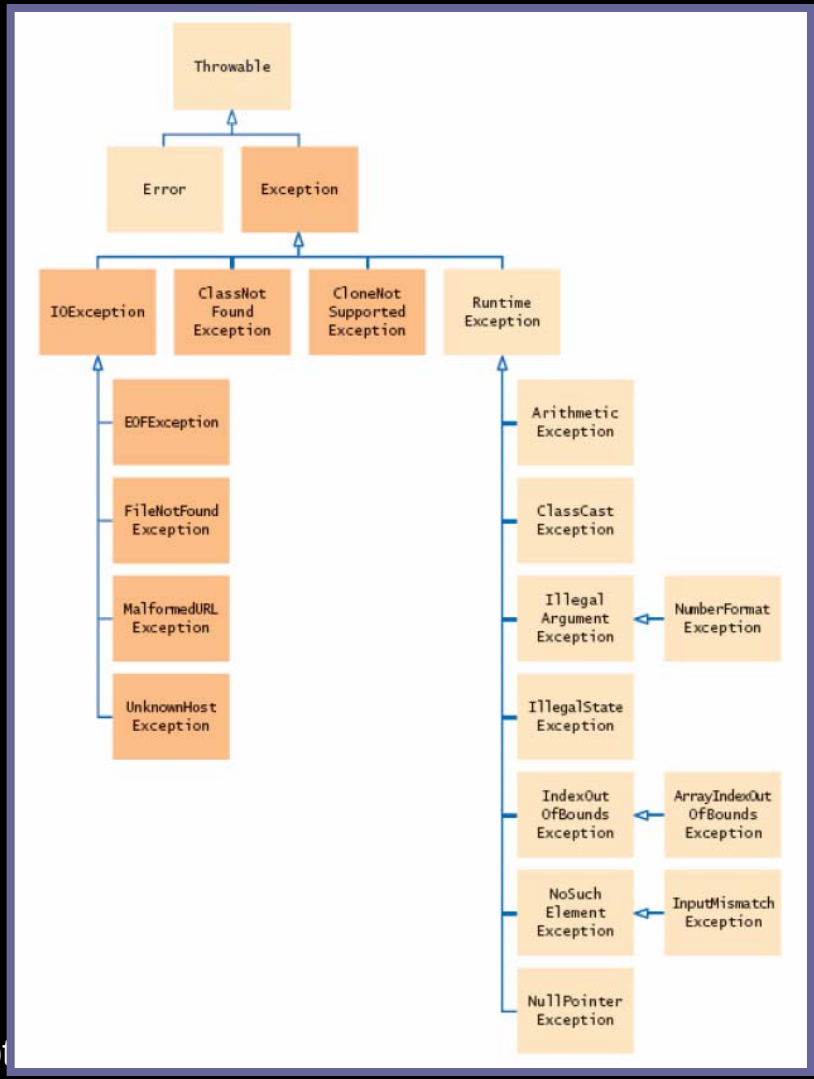

Figure 1:
The Hierarchy of Exception Classes

Syntax 15.1: Throwing an Exception

```
throw exceptionObject;
```

Example:

```
throw new IllegalArgumentException();
```

Purpose:

To throw an exception and transfer control to a handler for this exception type

Self Check

1. How should you modify the `deposit` method to ensure that the balance is never negative?
2. Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of `balance` afterwards?

Answers

1. Throw an exception if the amount being deposited is less than zero.
2. The balance is still zero because the last statement of the `withdraw` method was never executed.

Checked and Unchecked Exceptions

- **Two types of exceptions:**
  - Checked
    - The compiler checks that you don't ignore them
    - Due to external circumstances that the programmer cannot prevent
    - Majority occur when dealing with input and output
    - For example, `IOException`

Checked and Unchecked Exceptions

- **Two types of exceptions:**
  - Unchecked:
    - Extend the class `RuntimeException or Error`
    - They are the programmer's fault
    - Examples of runtime exceptions:

```
NumberFormatException
IllegalArgumentException
NullPointerException
```

    - Example of error: `OutOfMemoryError`

Checked and Unchecked Exceptions

- **Categories aren't perfect:**
  - `Scanner.nextInt` throws unchecked `InputMismatchException`
  - Programmer cannot prevent users from entering incorrect input
  - This choice makes the class easy to use for beginning programmers
- **Deal with checked exceptions principally when programming with files and streams**

Continued…

Checked and Unchecked Exceptions

- For example, use a `Scanner` to read a file

```
String filename = . . .;
FileReader reader = new FileReader(filename);
Scanner in = new Scanner(reader);
```

But, `FileReader` constructor can throw a `FileNotFoundException`

Checked and Unchecked Exceptions

- **Two choices:**
  - Handle the exception
  - Tell compiler that you want method to be terminated when the exception occurs
    - Use `throws` specifier so method can throw a checked exception

```
public void read(String filename) throws FileNotFoundException
{
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   . . .
}
```


Continued…

Checked and Unchecked Exceptions

- For multiple exceptions:

```
public void read(String filename)
      throws IOException, ClassNotFoundException
```

- Keep in mind inheritance hierarchy:
  If method can throw an `IOException` and `FileNotFoundException`, only use `IOException`

- **Better to declare exception than to handle it incompetently**

Syntax 15.2: Exception Specification

```
accessSpecifier returnType
      methodName(parameterType parameterName, . . .)
            throws ExceptionClass, ExceptionClass, . . .
```

Example:

```
public void read(BufferedReader in) throws IOException
```

Purpose:

To indicate the checked exceptions that this method can throw

Self Check

1. Suppose a method calls the `FileReader` constructor and the `read` method of the `FileReader` class, which can throw an `IOException`. Which `throws` specification should you use?
2. Why is a `NullPointerException` not a checked exception?

Answer

1. The specification throws `IOException` is sufficient because `FileNotFoundException` is a subclass of `IOException`.
2. Because programmers should simply check for `null` pointers instead of trying to handle a `NullPointerException`.

Catching Exceptions

- Install an exception handler with `try/catch` statement
- `try` block contains statements that may cause an exception
- `catch` clause contains handler for an exception type

Continued…

Catching Exceptions

- **Example:**

```
try
{
   String filename = . . .;
   FileReader reader = new FileReader(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
   catch (NumberFormatException exception)
{
   System.out.println("Input was not a number");
}
```

Catching Exceptions

- Statements in `try` block are executed
- If no exceptions occur, `catch` clauses are skipped
- If exception of matching type occurs, execution jumps to `catch` clause
- If exception of another type occurs, it is thrown until it is caught by another `try` block

Continued…

Catching Exceptions

- **`catch (IOException exception) block`**
  - `exception` contains reference to the exception object that was thrown
  - `catch` clause can analyze object to find out more details
  - `exception.printStackTrace()`: printout of chain of method calls that lead to exception

Syntax 15.3: General Try Block

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
. . .
```

Continued…

Syntax 15.3: General Try Block

Example:

```
 try
{
   System.out.println("How old are you?");
   int age = in.nextInt();
   System.out.println("Next year, you'll be " + (age + 1));
}
catch (InputMismatchException exception)
{
   exception.printStackTrace();
}
```

Purpose:

To execute one or more statements that may generate exceptions. If an exception occurs and it matches one of the catch clauses, execute the first one that matches. If no exception occurs, or an exception is thrown that doesn't match any catch clause, then skip the catch clauses.

Self Check

1. Suppose the file with the given file name exists and has no contents. Trace the flow of execution in the `try` block in this section.
2. Is there a difference between catching checked and unchecked exceptions?

Answers

1. The `FileReader` constructor succeeds, and in is constructed. Then the call `in.next()` throws a `NoSuchElementException`, and the `try` block is aborted. None of the `catch` clauses match, so none are executed. If none of the enclosing method calls catch the exception, the program terminates.

Continued…

Answers

2. No–you catch both exception types in the same way, as you can see from the code example on page 558. Recall that `IOException` is a checked exception and `NumberFormatException` is an unchecked exception.

The `finally` clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```
reader = new FileReader(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close();
// May never get here
```

The `finally` clause

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed "no matter what"

The `finally` clause

```
FileReader reader = new FileReader(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close(); // if an exception occurs, finally clause
                   // is also executed before exception is
                   // passed to its handler
}
```

The `finally` clause

- **Executed when `try` block is exited in any of three ways:**
  - After last statement of `try` block
  - After last statement of `catch` clause, if this `try` block caught an exception
  - When an exception was thrown in `try` block and not caught
- **Recommendation: don't mix `catch` and `finally` clauses in same `try` block**

Syntax 15.4: The `finally` clause

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

Continued…

Syntax 15.4: The `finally` clause

Example:

```
 FileReader reader = new FileReader(filename);
try
{
   readData(reader);
}
finally
{
   reader.close();
}
```

Purpose:

To ensure that the statements in the finally clause are executed whether or not the statements in the try block throw an exception.

Self Check

1. Why was the `reader` variable declared outside the `try` block?
2. Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.

Answers

1. If it had been declared inside the `try` block, its scope would only have extended to the end of the `try` block, and the catch clause could not have closed it.
2. The `FileReader` constructor throws an exception. The `finally` clause is executed. Since `reader` is `null`, the call to `close` is not executed. Next, a `catch` clause that matches the `FileNotFoundException` is located. If none exists, the program terminates.

Designing Your Own Execution Types

- You can design your own exception types–subclasses of `Exception` or `RuntimeException`
-

```
if (amount > balance)
{
   throw new InsufficientFundsException(
      "withdrawal of " + amount + " exceeds balance of
         " + balance);
}
```

- Make it an unchecked exception–programmer could have avoided it by calling `getBalance` first

Continued…

Designing Your Own Execution Types

- **Make it an unchecked exception– programmer could have avoided it by calling `getBalance` first**
- **Extend `RuntimeException` or one of its subclasses**
- **Supply two constructors**
  1. Default constructor
  2. A constructor that accepts a message string describing reason for exception

Designing Your Own Execution Types

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

Self Check

1. What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?
2. Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`. Which exception class should you extend?

Answers

1. To pass the exception message string to the `RuntimeException` superclass.
2. `Exception` or `IOException` are both good choices. Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException`.

A Complete Program

- **Program**
  - Asks user for name of file
  - File expected to contain data values
  - First line of file contains total number of values
  - Remaining lines contain the data
  - Typical input file:

```
3
1.45
 -2.1
0.05
```

A Complete Program

- **What can go wrong?**
  - File might not exist
  - File might have data in wrong format
- **Who can detect the faults?**
  - `FileReader` constructor will throw an exception when file does not exist
  - Methods that process input need to throw exception if they find error in data format

Continued…

A Complete Program

- **What exceptions can be thrown?**
  - `FileNotFoundException` can be thrown by `FileReader` constructor
  - `IOException` can be thrown by `close` method of `FileReader`
  - `BadDataException`, a custom checked exception class

Continued…

A Complete Program

- **Who can remedy the faults that the exceptions report?**
  - Only the `main` method of `DataSetTester` program interacts with user
    - Catches exceptions
    - Prints appropriate error messages
    - Gives user another chance to enter a correct file

File DataSetTester.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: public class DataSetTester
06: {
07:    public static void main(String[] args)
08:    {
09:       Scanner in = new Scanner(System.in);
10:       DataSetReader reader = new DataSetReader();
11:
12:       boolean done = false;
13:       while (!done)
14:       {
15:          try
16:          {
```

Continued...

File DataSetTester.java

```
17:             System.out.println("Please enter the file name: ");
18:             String filename = in.next();
19:
20:             double[] data = reader.readFile(filename);
21:             double sum = 0;
22:             for (double d : data) sum = sum + d;
23:             System.out.println("The sum is " + sum);
24:             done = true;
25:          }
26:          catch (FileNotFoundException exception)
27:          {
28:             System.out.println("File not found.");
29:          }
30:          catch (BadDataException exception)
31:          {
32:             System.out.println
                   ("Bad data: " + exception.getMessage());
```

Continued…

File DataSetTester.java

```
33:             }
34:             catch (IOException exception)
35:             {
36:                exception.printStackTrace();
37:             }
38:          }
39:       }
40: }
```

The `readFile` method of the `DataSetReader` class

- Constructs `Scanner` object
- Calls readData method
- Completely unconcerned with any exceptions

Continued…

The `readFile` method of the `DataSetReader` class

- If there is a problem with input file, it simply passes the exception to caller

```
public double[] readFile(String filename)
      throws IOException, BadDataException
      // FileNotFoundException is an IOException
{
   FileReader reader = new FileReader(filename);
   try
   {
      Scanner in = new Scanner(reader);
      readData(in);
   }
```

Continued…

The readFile method of the DataSetReader class

```
   finally
   {
      reader.close();
   }
   return data;
}
```

The `readFile` method of the `DataSetReader` class

- Reads the number of values
- Constructs an array
- Calls `readValue` for each data value

```
private void readData(Scanner in) throws BadDataException
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, i);

   if (in.hasNext())
      throw new BadDataException("End of file expected");
}
```

The `readFile` method of the `DataSetReader` class

- **Checks for two potential errors**
  1. File might not start with an integer
  2. File might have additional data after reading all values
- **Makes no attempt to catch any exceptions**

The readFile method of the DataSetReader class

```
private void readValue(Scanner in, int i)
      throws BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` calls `DataSetReader.readFile`
2. `readFile` calls `readData`
3. `readData` calls `readValue`
4. `readValue` doesn't find expected value and throws `BadDataException`
5. `readValue` has no handler for exception and terminates

Continued…

Scenario

1. `readData` has no handler for exception and terminates
2. `readFile` has no handler for exception and terminates after executing finally clause
3. `DataSetTester.main` has handler for `BadDataException`; handler prints a message, and user is given another chance to enter file name

File DataSetReader.java

```
01: import java.io.FileReader;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:    Reads a data set from a file. The file must have
          // the format
07:    numberOfValues
08:    value1
09:    value2
10:    . . .
11: */
12: public class DataSetReader
13: {
```

Continued…

File DataSetReader.java

```
14:    /**
15:       Reads a data set.
16:       @param filename the name of the file holding the data
17:       @return the data in the file
18:    */
19:    public double[] readFile(String filename)
20:          throws IOException, BadDataException
21:    {
22:       FileReader reader = new FileReader(filename);
23:       try
24:       {
25:          Scanner in = new Scanner(reader);
26:          readData(in);
27:       }
28:       finally
29:       {
30:          reader.close();
31:       }
```

Continued…

File DataSetReader.java

```
32:          return data;
33:       }
34:
35:       /**
36:          Reads all data.
37:          @param in the scanner that scans the data
38:       */
39:       private void readData(Scanner in) throws BadDataException
40:       {
41:          if (!in.hasNextInt())
42:             throw new BadDataException("Length expected");
43:          int numberOfValues = in.nextInt();
44:          data = new double[numberOfValues];
45:
46:          for (int i = 0; i < numberOfValues; i++)
47:             readValue(in, i);
```

Continued…

File DataSetReader.java

```
48:
49:          if (in.hasNext())
50:             throw new BadDataException("End of file expected");
51:       }
52:
53:       /**
54:          Reads one data value.
55:          @param in the scanner that scans the data
56:          @param i the position of the value to read
57:       */
58:       private void readValue(Scanner in, int i)
             throws BadDataException
59:       {
```

Continued…

File DataSetReader.java

```
60:          if (!in.hasNextDouble())
61:             throw new BadDataException("Data value expected");
62:          data[i] = in.nextDouble();
63:       }
64:
65:       private double[] data;
66: }
```

Self Check

1. Why doesn't the `DataSetReader.readFile` method catch any exceptions?
2. Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

Answers

1. It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

Continued…

Answers

2. `DataSetTester.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught.