Exception and Serialization

Han-fen Hu

Tony Gaddis (2019) Starting Out with Java: From Control Structures through Data Structures, 4th Edition

UNLV

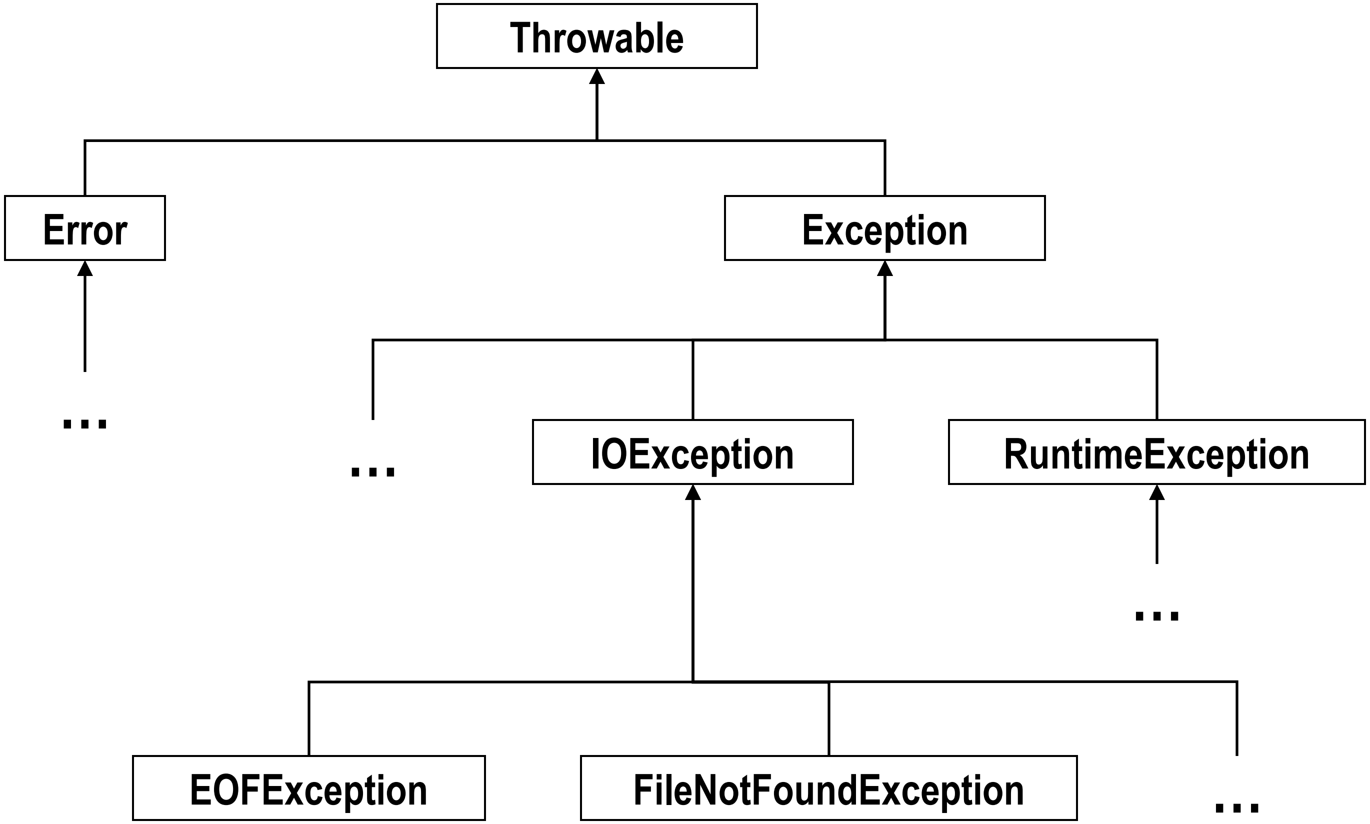
Outline

- Handling Exceptions
- Creating Exception Classes
- Binary Files and Object Serialization

Exception Classes

- An exception is an object that is generated as the result of an error or an unexpected event.
  - Exception are said to have been "thrown."
- Exception objects are created from classes in the Java API of Exception classes.
- All of the exception classes in the hierarchy are derived from the **`Throwable`** class.
  - **`Error`** and **`Exception`** are derived from the **`Throwable`** class.

Exception Classes in Java API

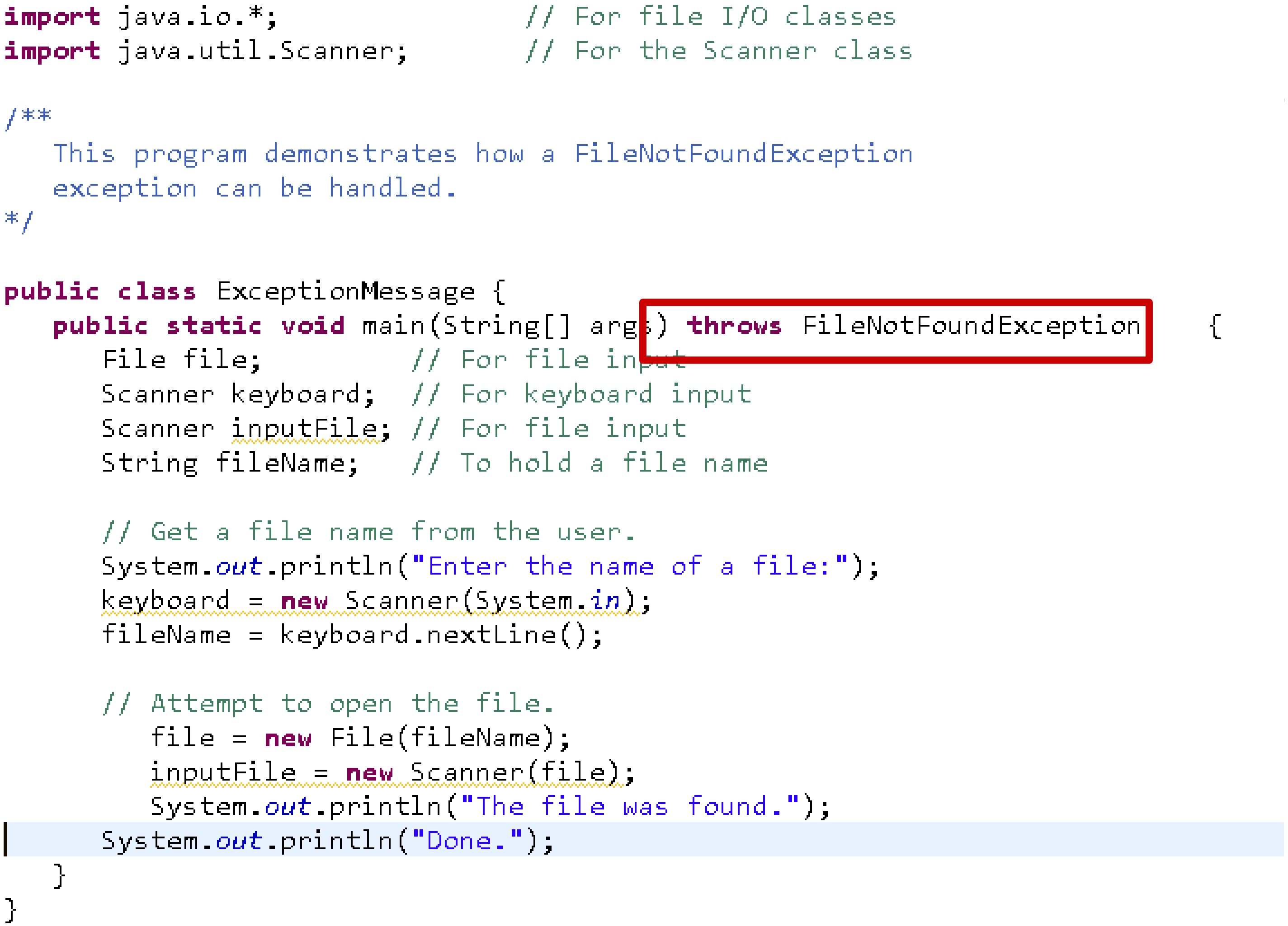


Error vs. Exception

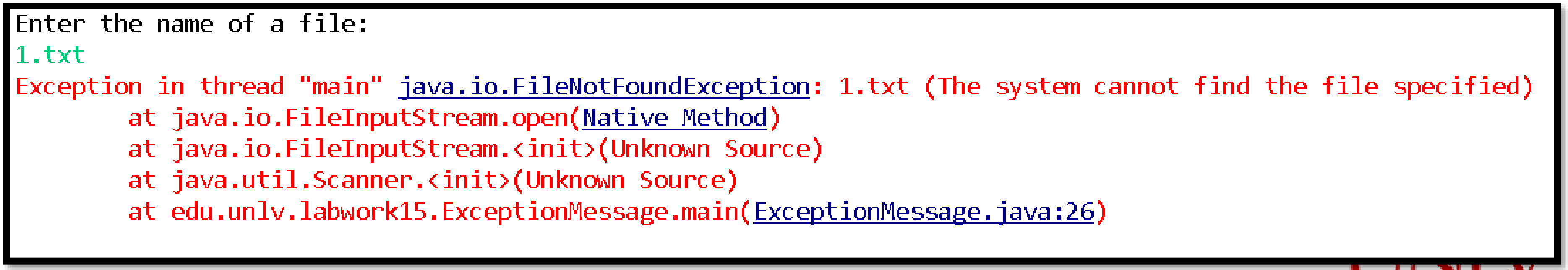
- Classes derived from **`Error`**:
  - Thrown when critical errors occur
    - An internal error in the Java Virtual Machine, or
    - Running out of memory
  - Critical issues that are out of the control of this program
  - Applications should not try to handle these errors because they are the result of a serious condition.
- Programmers should handle the exceptions from the **`Exception`** class
  - Issues that can be avoid with good program design
  - Special conditions that can be expected
  - Special cases that should be taken into account

Handling Exceptions (1)

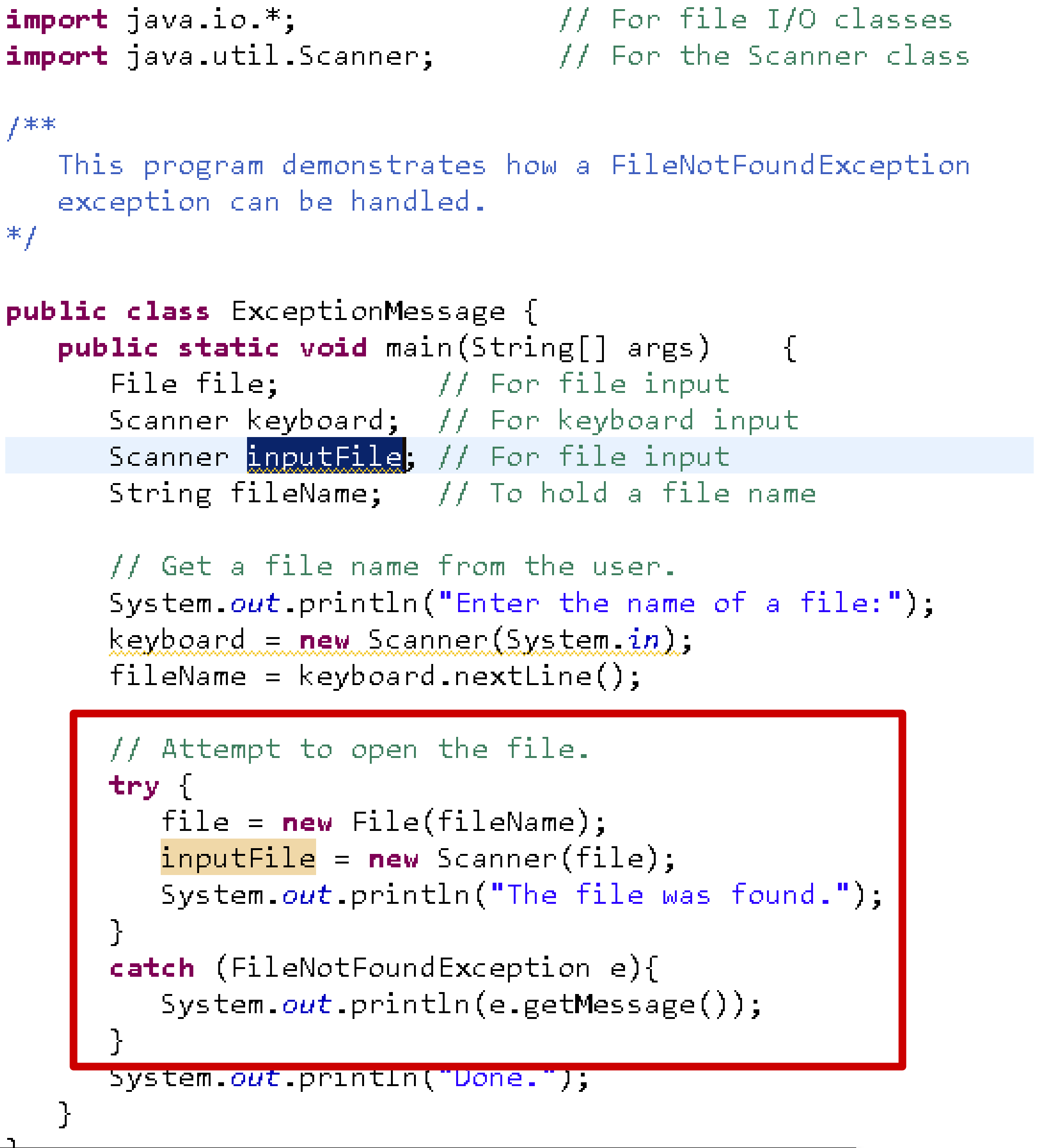
- Unhandled exceptions will crash a program.
- It is the programmers’ responsibility to write code that detects and handles exceptions.
  - Use the *default exception handler*
    - prints an error message and crashes the program
  - Create customized exception handlers
    - Respond to exceptions properly



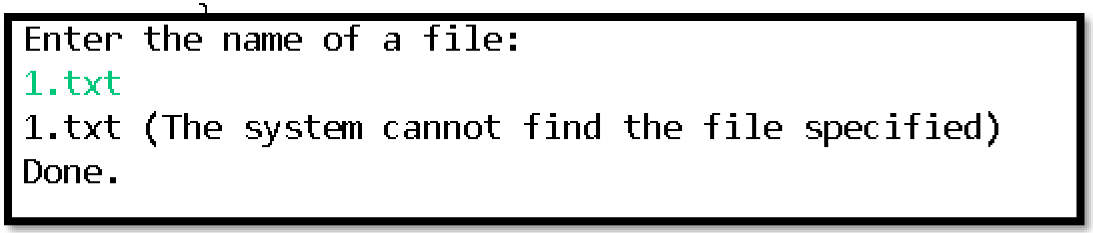
```java
import java.io.*;               // For file I/O classes
import java.util.Scanner;       // For the Scanner class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage {
   public static void main(String[] args) throws FileNotFoundException    {
      File file;          // For file input
      Scanner keyboard;   // For keyboard input
      Scanner inputFile;  // For file input
      String fileName;    // To hold a file name

      // Get a file name from the user.
      System.out.println("Enter the name of a file:");
      keyboard = new Scanner(System.in);
      fileName = keyboard.nextLine();

      // Attempt to open the file.
         file = new File(fileName);
         inputFile = new Scanner(file);
         System.out.println("The file was found.");
      System.out.println("Done.");
   }
}
```



```
Enter the name of a file:
1.txt
Exception in thread "main" java.io.FileNotFoundException: 1.txt (The system cannot find the file specified)
	at java.io.FileInputStream.open(Native Method)
	at java.io.FileInputStream.<init>(Unknown Source)
	at java.util.Scanner.<init>(Unknown Source)
	at edu.unlv.labwork15.ExceptionMessage.main(ExceptionMessage.java:26)
```



```java
import java.io.*;             // For file I/O classes
import java.util.Scanner;     // For the Scanner class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage {
   public static void main(String[] args)    {
      File file;          // For file input
      Scanner keyboard;   // For keyboard input
      Scanner inputFile;  // For file input
      String fileName;    // To hold a file name

      // Get a file name from the user.
      System.out.println("Enter the name of a file:");
      keyboard = new Scanner(System.in);
      fileName = keyboard.nextLine();

      // Attempt to open the file.
      try {
         file = new File(fileName);
         inputFile = new Scanner(file);
         System.out.println("The file was found.");
      }
      catch (FileNotFoundException e){
         System.out.println(e.getMessage());
      }
      System.out.println("Done.");
   }
}
```



```
Enter the name of a file:
1.txt
1.txt (The system cannot find the file specified)
Done.
```

Handling Exceptions (2)

- An *exception handler* is a section of code that gracefully responds to exceptions
  - The process of intercepting and responding to exceptions is called *exception handling*.
- The *default exception handler* deals with unhandled exceptions

try-catch Statement (1)

```
try{
   (try block statements...)
}
catch (ExceptionType ParameterName){
   (catch block statements...)
}
```

- try clause indicates a block of code will be attempted (the curly braces are required)
  - One or more statements that are executed and can potentially throw an exception
  - The application will not halt if the try block throws an exception

try-catch Statement (2)

- catch clause appears immediately after try block (the curly braces are required)
  - ***ExceptionType*** is the name of an exception class
  - ***ParameterName*** is a variable name which will reference the exception object thrown by the *try* block
  - The code in the catch block is executed if the try block throws an exception.

try-catch Statement: Example

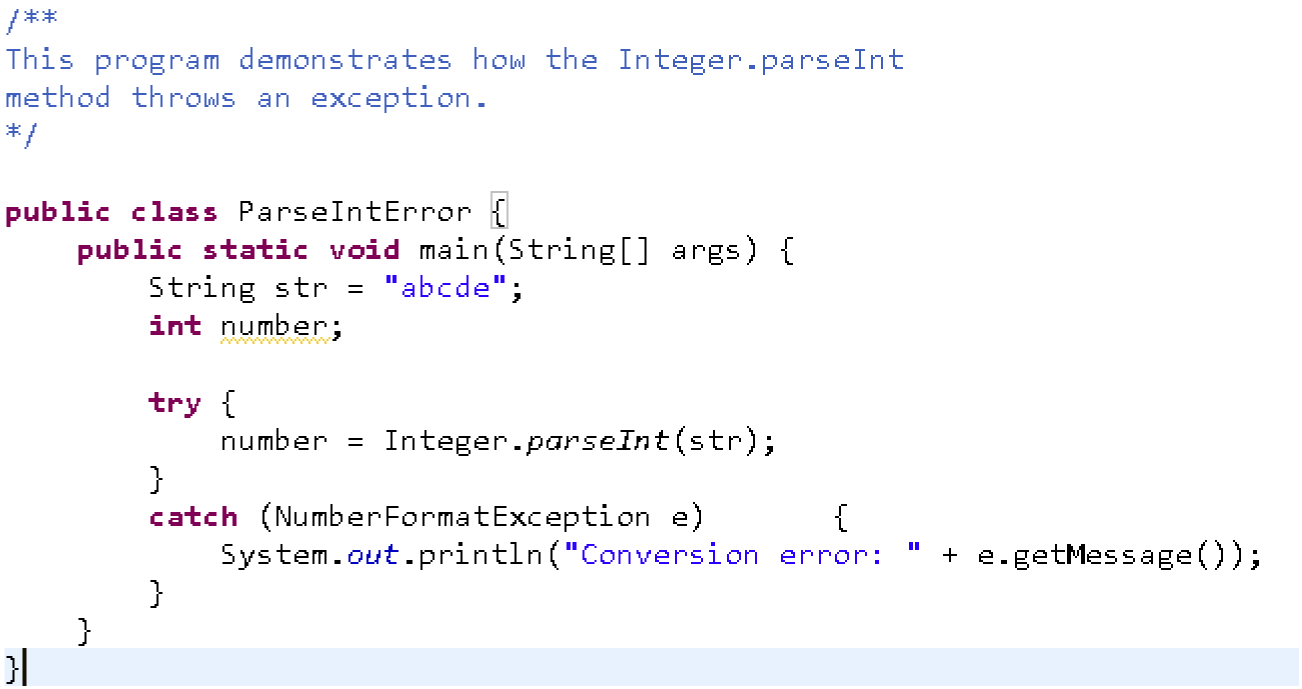
- This code is designed to handle a **`FileNotFoundException`** if it is thrown.

```
try{
  File file = new File ("MyFile.txt");
  Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e){
  System.out.println("File not found.");
}
```

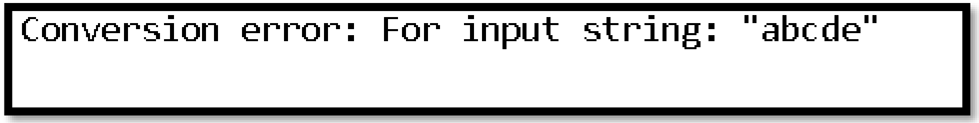
- The parameter must be of a type that is compatible with the thrown exception’s type.
- After an exception, the program will continue execution

Exception Message

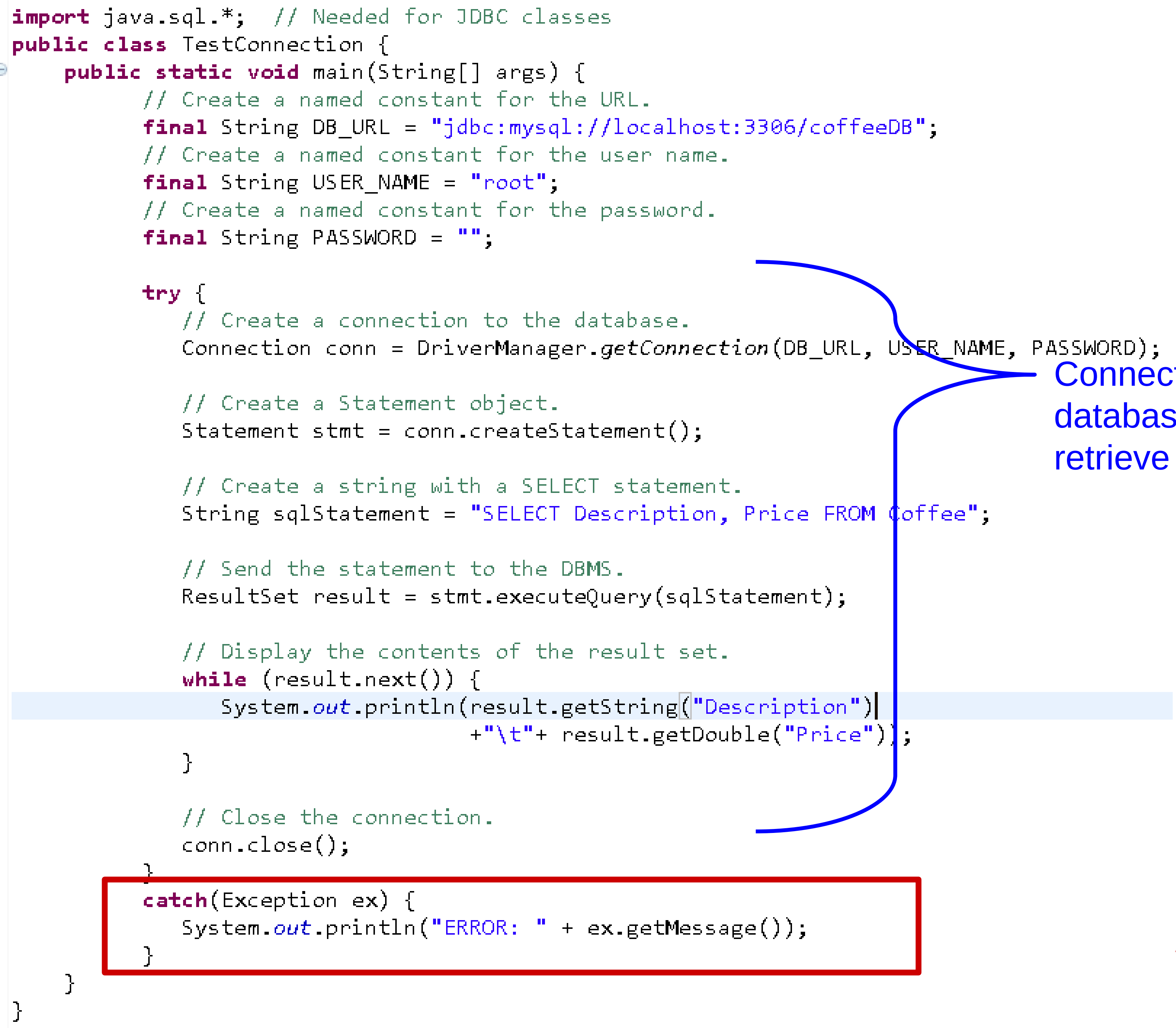
- Each exception object has a method named **getMessage** that can be used to retrieve the default error message for the exception.



```
/**
This program demonstrates how the Integer.parseInt
method throws an exception.
*/

public class ParseIntError {
    public static void main(String[] args) {
        String str = "abcde";
        int number;

        try {
            number = Integer.parseInt(str);
        }
        catch (NumberFormatException e)       {
            System.out.println("Conversion error: " + e.getMessage());
        }
    }
}
```

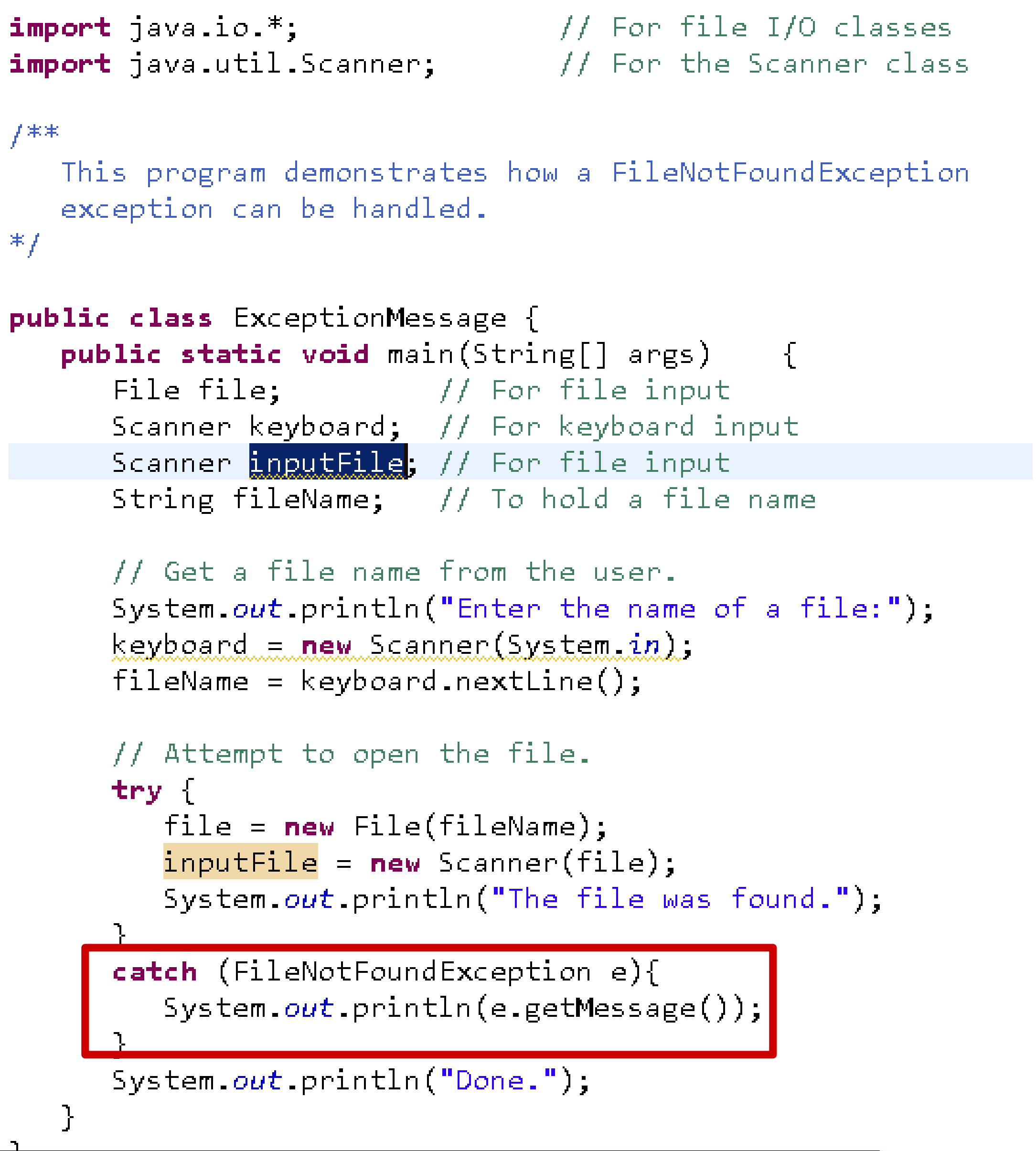


```
Conversion error: For input string: "abcde"
```

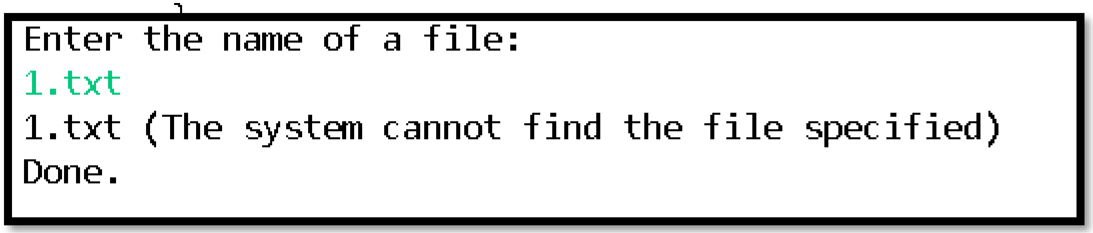


```java
import java.sql.*;  // Needed for JDBC classes
public class TestConnection {
    public static void main(String[] args) {
          // Create a named constant for the URL.
          final String DB_URL = "jdbc:mysql://localhost:3306/coffeeDB";
          // Create a named constant for the user name.
          final String USER_NAME = "root";
          // Create a named constant for the password.
          final String PASSWORD = "";

          try {
             // Create a connection to the database.
             Connection conn = DriverManager.getConnection(DB_URL, USER_NAME, PASSWORD);

             // Create a Statement object.
             Statement stmt = conn.createStatement();

             // Create a string with a SELECT statement.
             String sqlStatement = "SELECT Description, Price FROM Coffee";

             // Send the statement to the DBMS.
             ResultSet result = stmt.executeQuery(sqlStatement);

             // Display the contents of the result set.
             while (result.next()) {
                System.out.println(result.getString("Description")
                                      +"\t"+ result.getDouble("Price"));
             }

             // Close the connection.
             conn.close();
          }
          catch(Exception ex) {
             System.out.println("ERROR: " + ex.getMessage());
          }
    }
}
```

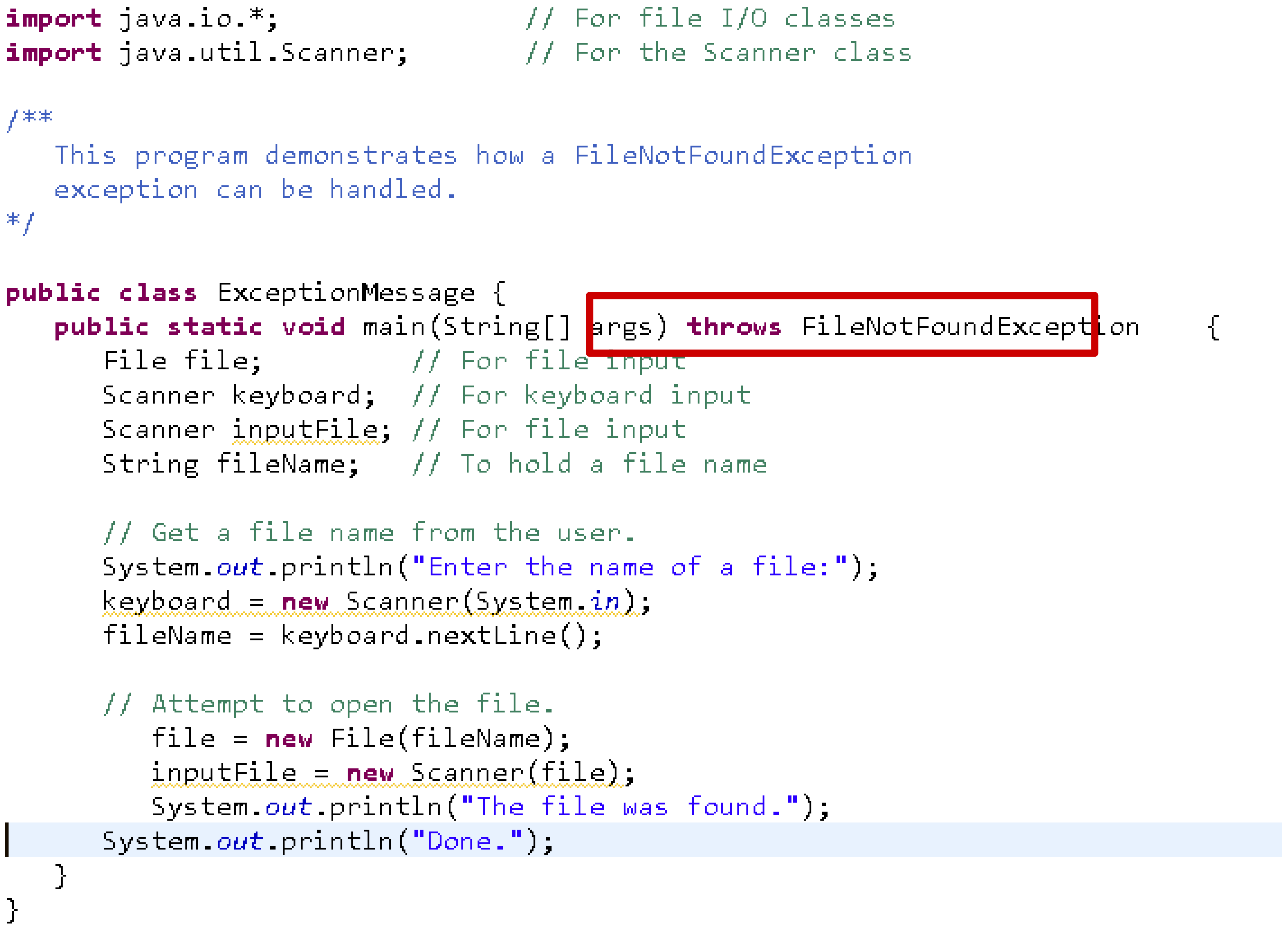
Connect to the database and retrieve data



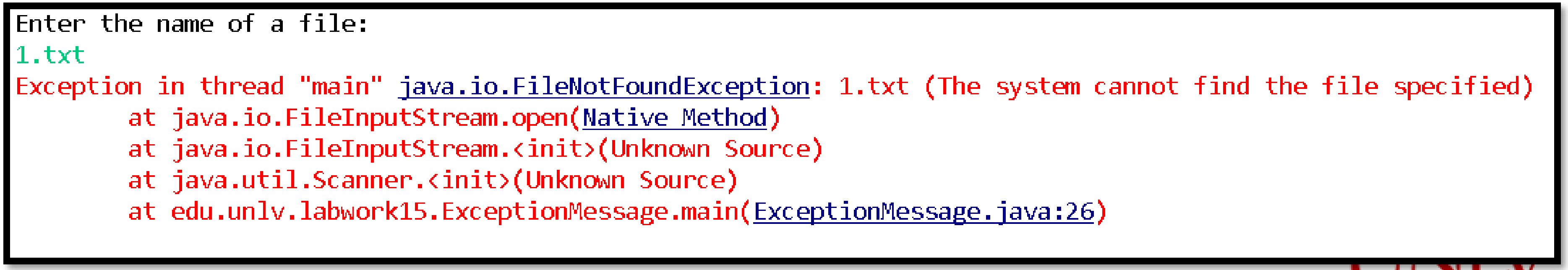
```java
import java.io.*;             // For file I/O classes
import java.util.Scanner;     // For the Scanner class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage {
   public static void main(String[] args)    {
      File file;            // For file input
      Scanner keyboard;     // For keyboard input
      Scanner inputFile;    // For file input
      String fileName;      // To hold a file name

      // Get a file name from the user.
      System.out.println("Enter the name of a file:");
      keyboard = new Scanner(System.in);
      fileName = keyboard.nextLine();

      // Attempt to open the file.
      try {
         file = new File(fileName);
         inputFile = new Scanner(file);
         System.out.println("The file was found.");
      }
      catch (FileNotFoundException e){
         System.out.println(e.getMessage());
      }
      System.out.println("Done.");
   }
}
```



```
Enter the name of a file:
1.txt
1.txt (The system cannot find the file specified)
Done.
```



```java
import java.io.*;           // For file I/O classes
import java.util.Scanner;   // For the Scanner class

/**
   This program demonstrates how a FileNotFoundException
   exception can be handled.
*/

public class ExceptionMessage {
   public static void main(String[] args) throws FileNotFoundException    {
      File file;          // For file input
      Scanner keyboard;   // For keyboard input
      Scanner inputFile;  // For file input
      String fileName;    // To hold a file name

      // Get a file name from the user.
      System.out.println("Enter the name of a file:");
      keyboard = new Scanner(System.in);
      fileName = keyboard.nextLine();

      // Attempt to open the file.
         file = new File(fileName);
         inputFile = new Scanner(file);
         System.out.println("The file was found.");
      System.out.println("Done.");
   }
}
```



```
Enter the name of a file:
1.txt
Exception in thread "main" java.io.FileNotFoundException: 1.txt (The system cannot find the file specified)
	at java.io.FileInputStream.open(Native Method)
	at java.io.FileInputStream.<init>(Unknown Source)
	at java.util.Scanner.<init>(Unknown Source)
	at edu.unlv.labwork15.ExceptionMessage.main(ExceptionMessage.java:26)
```

Lab (1)

- SalesReport.java
  - In this program, a file contains sales data will be read and the sales average will be calculated.
  - It is subject to the potential issue of “file not found”

Exception Handling for Input Values

- For Scanner class, the nextXXX() method would fail if the input value does not match the type expected
- For wrapper classes, parse methods would fail if the input value does not match the type expected
- An exception will be thrown

Lab (2)

- InputValidationDemo.java
- InputValidationParseDemo.java

Handling Multiple Exceptions

- The code in the try block may be capable of throwing more than one type of exception.
- A catch clause needs to be written for each type of exception that could potentially be thrown.
- The JVM will run the first compatible catch clause found.

Lab (3)

- SalesReport.java
  - This program is subject to another issue: the data in the file are not numeric values.
  - Thus, another type of exception should also be handled.

Multiple Exception Handlers (1)

- A try statement may have only one **`catch`** clause for each specific type of exception.

```
try {
   number = Integer.parseInt(str);
}
catch (NumberFormatException e){
   System.out.println("Bad number format.");
}
catch (NumberFormatException e){ // ERROR!!!
   System.out.println(str + " is not a
  number.");
}
```

Multiple Exception Handlers (2)

- The `NumberFormatException` class is derived from the `IllegalArgumentException` class.
- The `catch` clauses must be listed from most specific to most general.

```
try {
    number = Integer.parseInt(str);
}
catch (IllegalArgumentException e) {
    System.out.println("Bad number format.");
}
catch (NumberFormatException e) {// ERROR!!!
    System.out.println(str + " is not a
  number.");
}
```

Multiple Exception Handlers (3)

- The previous code could be rewritten to work, as follows, with no errors:

```
try {
   number = Integer.parseInt(str);
}
catch (NumberFormatException e){
   System.out.println(str +" is not a number.");
}
catch (IllegalArgumentException e) { //OK
   System.out.println("Bad number format.");
}
```

The `finally` Clause

- The try statement may have an optional `finally` clause.
- If present, the `finally` clause must appear after all of the `catch` clauses.

```
try {
    (try block statements...)
} catch (ExceptionType ParameterName) {
    (catch block statements...)
} finally {
    (finally block statements...)
}
```

- The *finally block* is one or more statements that are always executed whether an exception occurs or not

Lab (4)

- SalesReportFinally.java
  - Once a file is opened, we need to close it. In the program, we will use a finally clause to close the file.
  - In this program, you will also see the example of nested exception handling

Multi-Catch

- You can also specify more than one exception in a **`catch`** clause:

```
try{
}
catch(NumberFormatException | InputMismatchException ex){
}
```

Separate the exceptions with the | character.

Lab (5)

- SalesReportMultiCatch.java
  - In this program, we intend to catch both FileNotFoundException and InputMismatchException with on catch clause.

Creating Exception Classes

- Create your own exception classes by deriving them from the `Exception` class or one of its derived classes
  - Handle specific business rules
  - Provide customized exception messages
- The constructor of `Exception` class accepts a string as the error message.
  - In the customized exception class derived from the `Exception` class, we need to also specify the error message by calling the super class constructor
- The constructor of the customized exception class can have parameters, or not.

Lab (6)

- BankAccount.java
  - An examples of exceptions that can affect a bank account: A negative balance is passed to the constructor
- NegativeStartingBalance.java
  - Exceptions that represent the error condition
- AccountDemo.java
  - Application class

`@exception` Tag

- In the code documentation, we can use the **`@exception`** tag to describe the exception handled.
- General format

  `@exception` *ExceptionName Description*

- The following rules apply
  - The **`@exception`** tag in a method's documentation comment must appear after the general description of the method.
  - The description can span several lines. It ends at the end of the documentation comment (the `*/` symbol) or at the beginning of another tag.

Lab (7)

- BankAccount.java
  - Add the method documentation of @exception to the constructor

Creating Exception Classes (2)

- Some other examples of exceptions that can affect a bank account:
  - A negative number is passed to the deposit method.
  - A negative number is passed to the withdraw method.
  - The amount passed to the withdraw method exceeds the account's balance.
- We can create exceptions that represent each of these error conditions.

Lab (8)

- BankAccount.java
  - Add throws clauses to deposit() and withdraw()
- NegativeAmount.java
  - New exception class
- Overdraft.java
  - New exception class
- BankAccountDemo2.java

File Streams

- Java views each file as a sequential stream of bytes
  - Character-based streams
    - Input and output data as a sequence of characters
    - Value 5 is stored as 5
  - Byte-based streams
    - Input and output data in the binary format.
    - Value 5 is stored as 101

Binary Files (1)

- A file that contains binary data is often called a binary file.
  - Cannot be opened in a text editor such as Notepad
  - More efficient than storing it as text (character-based).
- Some types of data that should only be stored in its raw binary format.
  - Keeping the content of objects
  - When the instance variables were output to a text file, certain information was lost, such as the type of each value (i.e., all values reading from a text file are strings.)

Object Serialization (1)

- If an object contains other types of objects as fields, saving its contents can be complicated.

- Serializing objects is a simpler way of saving objects to a file
  - When an object is serialized, it is converted into a series of bytes that contain the object’s data
  - The resulting set of bytes can be saved to a file for later retrieval

Object Serialization (2)

- For an object to be serialized, its class must implement the **`Serializable`** interface.
  - The **`Serializable`** interface has no methods or fields.
  - It is used only to let the Java compiler know that objects of the class might be serialized.
- If a class contains objects of other classes as fields, those classes must also implement the **`Serializable`** interface, in order to be serialized.

Object Serialization (3)

- Example

```
FileOutputStream outStream = new
   FileOutputStream("Objects.dat");

ObjectOutputStream objectOutputFile =
   new ObjectOutputStream(outStream);
```

Object Serialization (4)

- To serialize an object and write it to the file, the **`writeObject`** method is used.

```
BankAccount account = new BankAccount(25000.0);
objectOutputFile.writeObject(account);
```

  - The **`writeObject`** method throws an **`IOException`** if an error occurs.

Lab (9)

- BankAccount.java
- SerializeObjects.java
  - We will try to create an array on BankAccount and write the objects to a file

Object Serialization (5)

- The process of reading a serialized object's bytes and constructing an object from them is known as deserialization.
  - An **`ObjectInputStream`** object is used in conjunction with a **`FileInputStream`** object.

```
FileInputStream inStream = new
            FileInputStream("Objects.dat");
ObjectInputStream objectInputFile = new
            ObjectInputStream(inStream);
```

Object Serialization (6)

- To read a serialized object from the file, the **`readObject`** method is used.

```
BankAccount account;
account = (BankAccount)
                objectInputFile.readObject();
```

  - **`readObject`** method returns the deserialized object
    - Notice that you must cast the return value to the desired class type.
  - The **`readObject`** method throws a number of different exceptions if an error occurs.

Lab (10)

- BankAccount.java
- SerializeObjects.java
- DeserializeObjects.java