COP 3330: Object-Oriented Programming Summer 2007

Exception Handing in Java – Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

Exception Handling in Java

- An **exception** is an abnormal event that occurs during program execution.
  - attempt to manipulate a nonexistent file.
  - improper array subscripting.
  - improper arithmetic operations such as divide by zero.
- If an exception occurs and an **exception-handler** code segment is in effect for that exception, then flow of control is transferred to the handler.
- If an exception occurs and there is no handler for it, the program terminates.

Exception Handling in Java (cont.)

```
import java.io.*;
public class except_A {
   public static void main (String[] args) throws IOException {
     //get filename
     BufferedReader stdin = new BufferedReader(
               new InputStreamReader(System.in));
     System.out.println("Filename: ");
     String s = stdin.readLine();
     //set up file stream for processing
     BufferedReader filein = new BufferedReader(new FileReader(s));
     //extract values and perform calculation
     int numerator = Integer.parseInt(filein.readLine());
     int denominator = Integer.parseInt(filein.readLine());
     int quotient = numerator / denominator;
     System.out.println();
     System.out.println(numerator + " / " + denominator + " = " +
            quotient);
     return;
   }
}
```


Exception Handling in Java (cont.)

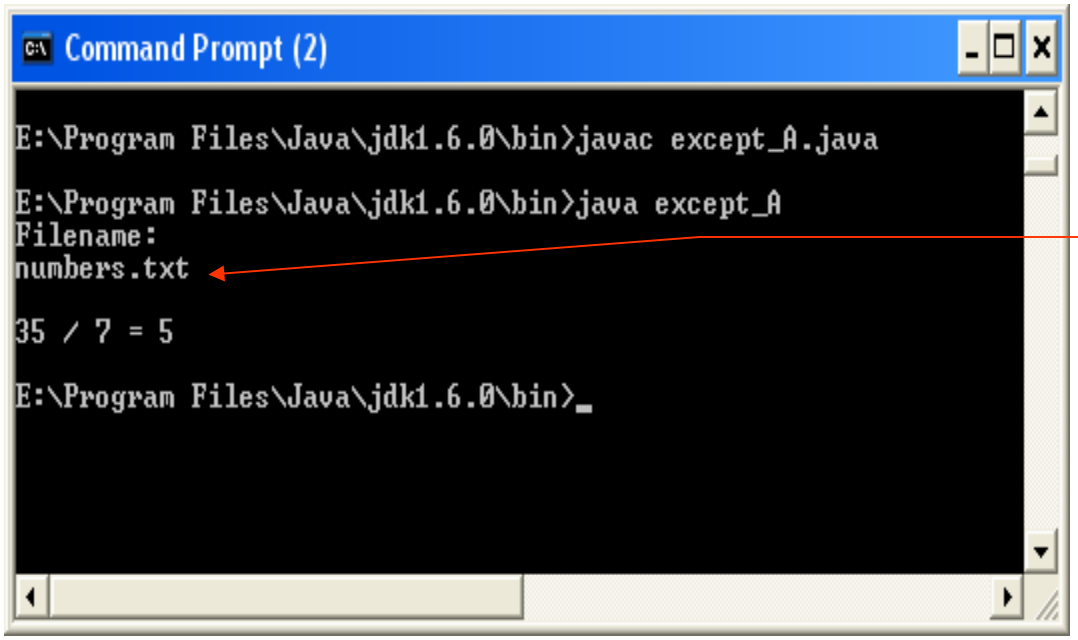
- There is a **throws** expression in the `main` method signature for class `except_A` shown in the previous slide.
  - All of the interactive console application programs that have appeared in the notes this semester have included throws expressions in their `main` method signatures.
- Java requires the throws expression for any method that does not handle the I/O exceptions that it may generate.

Exception Handling in Java (cont.)

- The inclusion of the throws expression is a warning to users of the method. Such knowledge is important because if an invoked method does not handle an exception, then the exception is given to the invoking method to handle.
  - If the invoking method does not handle the exception, then the unwinding process continues with the method that did the invoking of the invoking method, and so on. If no method is found in the unwinding process to handle the exception, then the program terminates.

Exception Handling in Java (cont.)

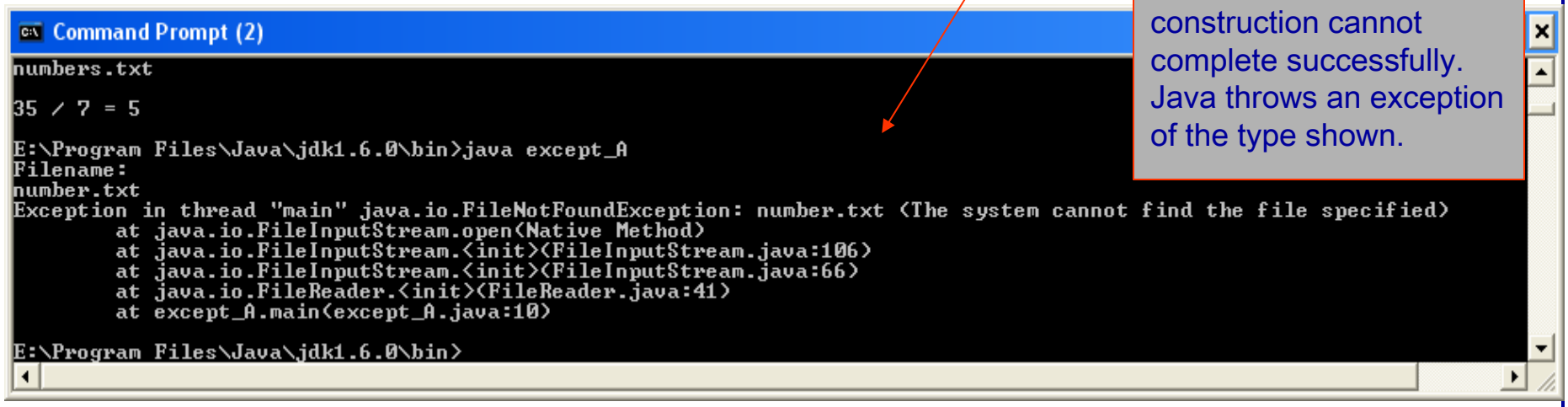
- Suppose that program `except_A` is executed and the user specifies the file to be the file named `numbers.txt` containing the values 35 and 7 on successive lines. The I/O behavior of the program is shown below.



User supplies a valid filename causing the quotient to be calculated and displayed.

Exception Handling in Java (cont.)

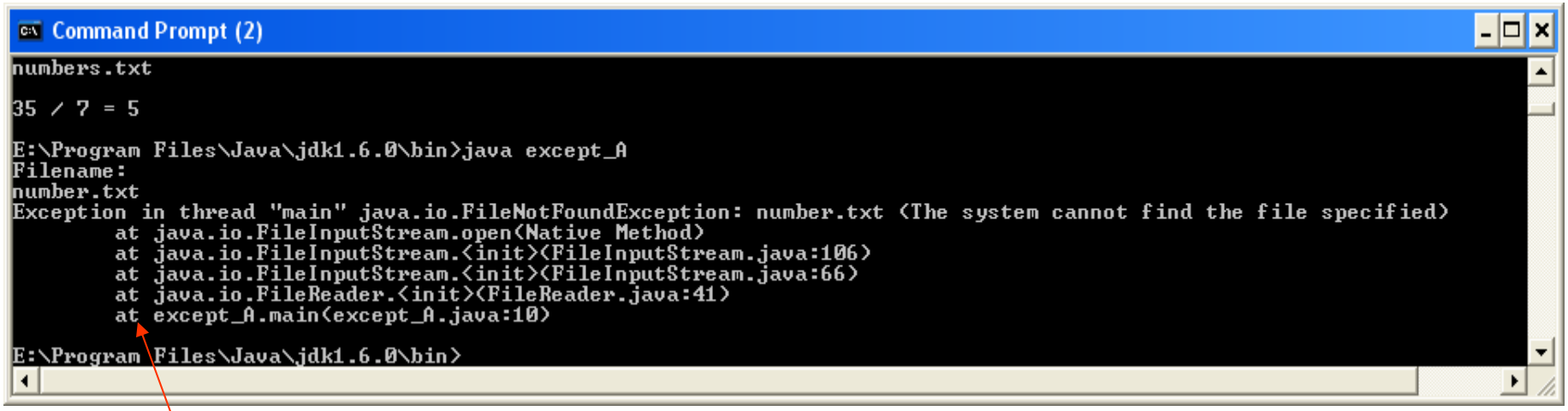
- What happens when program `except_A` is executed and the user specifies an invalid file? Suppose the user misspells the name of the original file as `number.txt`. The I/O behavior of the program for this execution is shown below.



Exception Handling in Java (cont.)

- When an exception is thrown and not handled, Java generates a message to the standard error stream indicating where the exception occurred.
  - By default, the standard error stream is the terminal screen.
- When examining an exception message, it is sometimes easier to start with the last line of the message and work your way toward the first message line. The last line indicates the start of the process that caused the throwing of the exception.

Exception Handling in Java (cont.)



Start here:

1. This line indicates that the exception was generated at line 10 in except.A.main. This line defines a BufferedReader variable filein.
2. In creating the FileReader used by the BufferedReader constructor,a FileInputStream was needed.
3. This FileInputStream used a method open() that interacted with the file system. Because of the misspelled name, the file could not be opened.
4. The exception reporting this problem was generated and eventually propagated to the main() method. Since main() did not handle this exception, the program was terminated.

Exception Handling in Java (cont.)

- In the preceding example, the program did not end gracefully. A user does not want to read the jargon of an exception message.
- A better alternative is to used Java's **`try-catch`** mechanism.
- With this mechanism, code that deals with situations in which exceptions can arise is put into a **`try`** block. If an exception arises in a `try` block, Java will transfer control to the appropriate exception handler to handle the problem.
  - A **`try`** block is a statement block with the keyword try preceding it.

Exception Handling in Java (cont.)

- The exception handlers are **`catch`** blocks and they immediately follow the `try` block.
- A **`catch`** block is a statement block that begins with the keyword `catch` followed by a single parameter that specifies the type of exception to be handled by the block.
- There is usually a `catch` handler for each type of exception that can occur within the try block.
- The `catch` blocks are executed only if an exception is generated.

Exception Handling in Java (cont.)

- General form of the `try-catch` mechanism:

```
try {Action} catch (ExceptionType Parameter) {Handler}
```

Block of code that can generate an exception of type ExceptionType

Type of exception to be caught

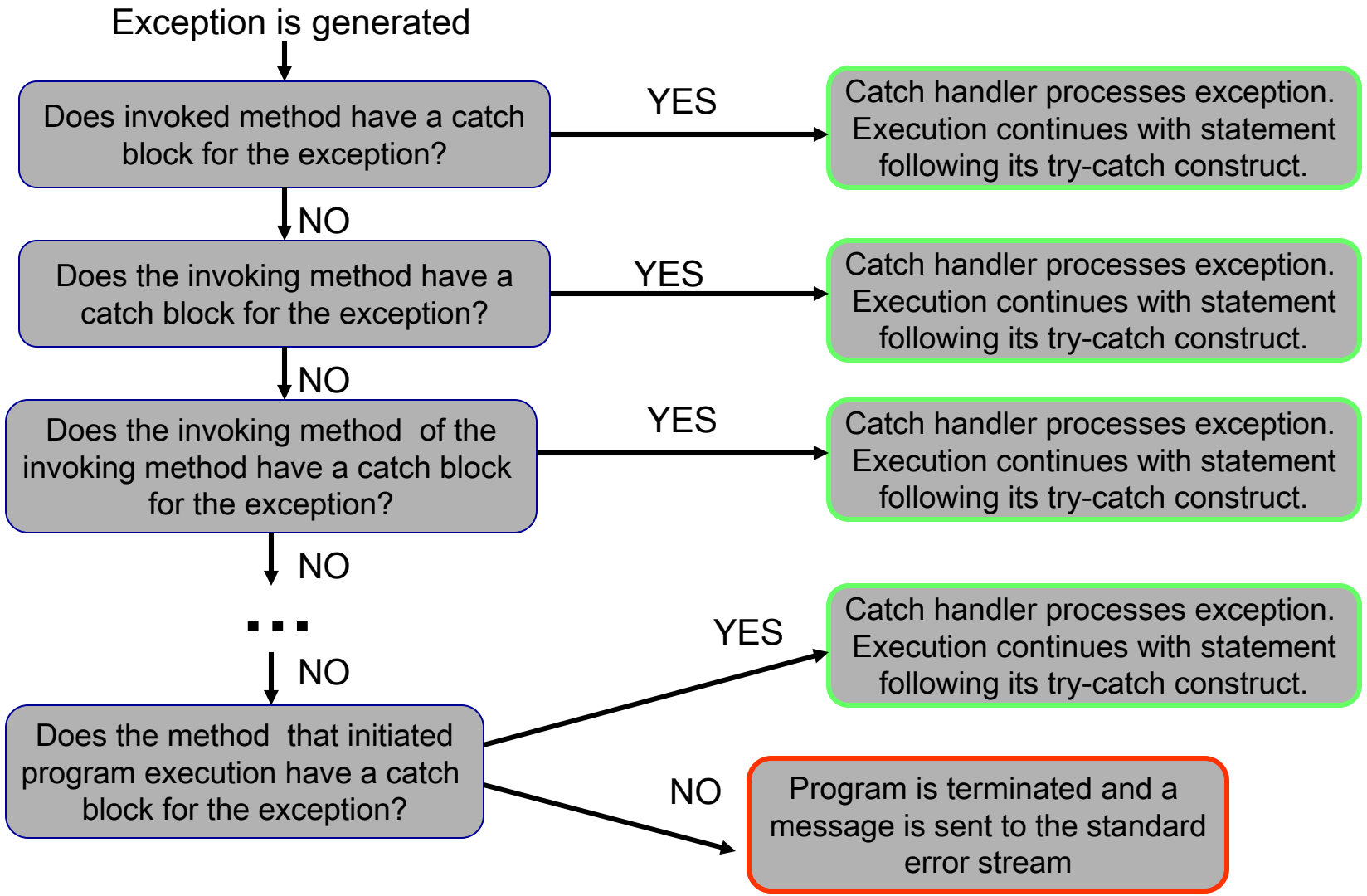
If an ExceptionType is generated, then Parameter is initialized with exception information

Block of code that handles the caught exception

Exception Handling in Java (cont.)

- If an exception is thrown in a try block, then the first associated catch block whose parameter type matches the generated exception is used.
- If no catch block parameter matches the exception, then the invocation process is unwound automatically to find the appropriate exception handler. The unwinding process is illustrated in the next slide.

Exception Handling Process



Exception Handling in Java (cont.)

- If a `catch` block is executed, then its parameter is initialized with information regarding the specifics of the exception.
- If the `catch` block does not end the program, then after the `catch` block completes, program execution continues with the statement following the `try-catch` construct.

Exception Handling in Java (cont.)

- The following code segment appears in an upgraded version of program except_A called except_B (pages 17-18) and catches the exception generated by an invalid filename.

```
//set up file stream for processing
BufferedReader filein = null;
try {
     filein = new BufferedReader(new FileReader(s));
}
catch (FileNotFoundException e) {
 System.err.println(s + ": cannot be opened for reading");
 System.exit(0);
}
```

Note: Variable filein is declared outside of the try-catch block.

Exception Handling in Java (cont.)

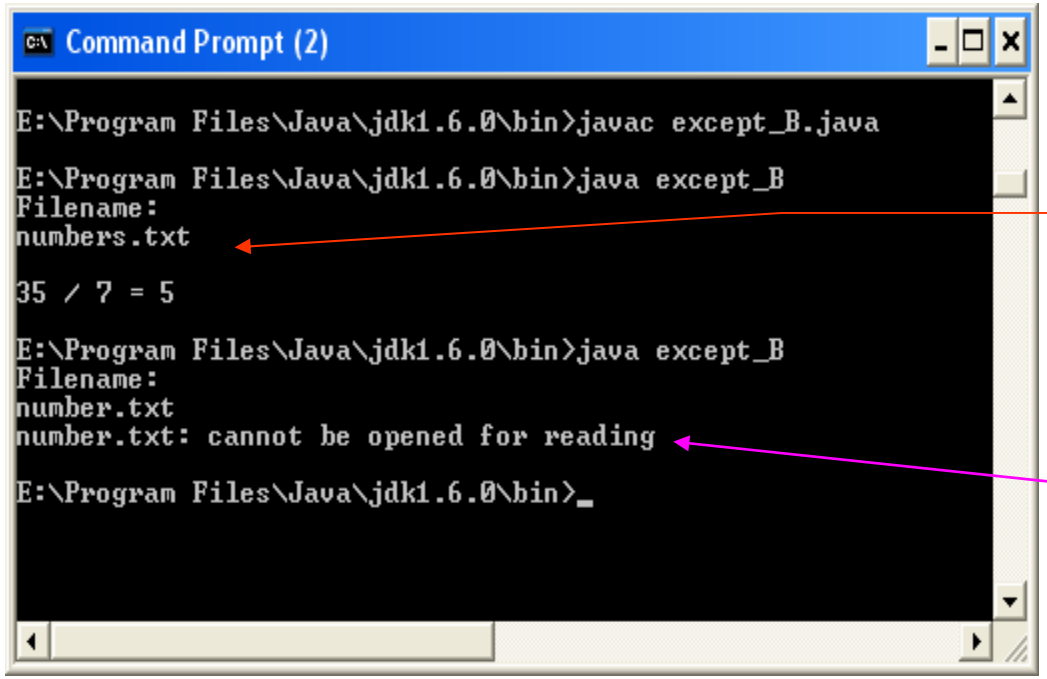
```
import java.io.*;

public class except_B {
   public static void main (String[] args) throws IOException {
     //get filename
     BufferedReader stdin = new BufferedReader(
               new InputStreamReader(System.in));
     System.out.println("Filename: ");
     String s = stdin.readLine();
     //set up file stream for processing
     BufferedReader filein = null;
     try {
          filein = new BufferedReader(new FileReader(s));
     }
     catch (FileNotFoundException e) {
          System.err.println(s + ": cannot be opened for reading");
          System.exit(0);
     }
```


Exception Handling in Java (cont.)

```
      //extract values and perform calculation
      int numerator = Integer.parseInt(filein.readLine());
      int denominator = Integer.parseInt(filein.readLine());
      int quotient = numerator / denominator;
      System.out.println();
      System.out.println(numerator + " / " + denominator + " = " +
            quotient);
      return;
   }
}
```


Exception Handling in Java (cont.)

- Suppose that program `except_B` is executed and the user specifies the file to be the file named `numbers.txt` containing the values 35 and 7 on successive lines. The I/O behavior of the program is shown below.



User supplies a valid filename causing the quotient to be calculated and displayed.

User enters an invalid filename causing an exception which is caught and handled.

Exception Handling in Java (cont.)

- Notice that in program `except_B` that the `main()` method contains a `throws` expression even though we have a try-catch construct for the `FileNotFoundException`. WHY?
  - Answer: It is still possible for the program to throw an IOException. In particular any of the `readLine()` invocations in the main method could generate an IOException if there is a file-system failure.

Exception Handling in Java (cont.)

- To remove the necessity of have the throws expression as part of the main method signature, the `readLine()` invocations will need to be within `try` blocks.
- Program `except_C` on the next page wraps the `readLine()` invocations into two `try` blocks. One for the `readLine()` invocation that is reading the filename and the second for the `readLine()` invocation that is reading in the numerator and denominator from the file.

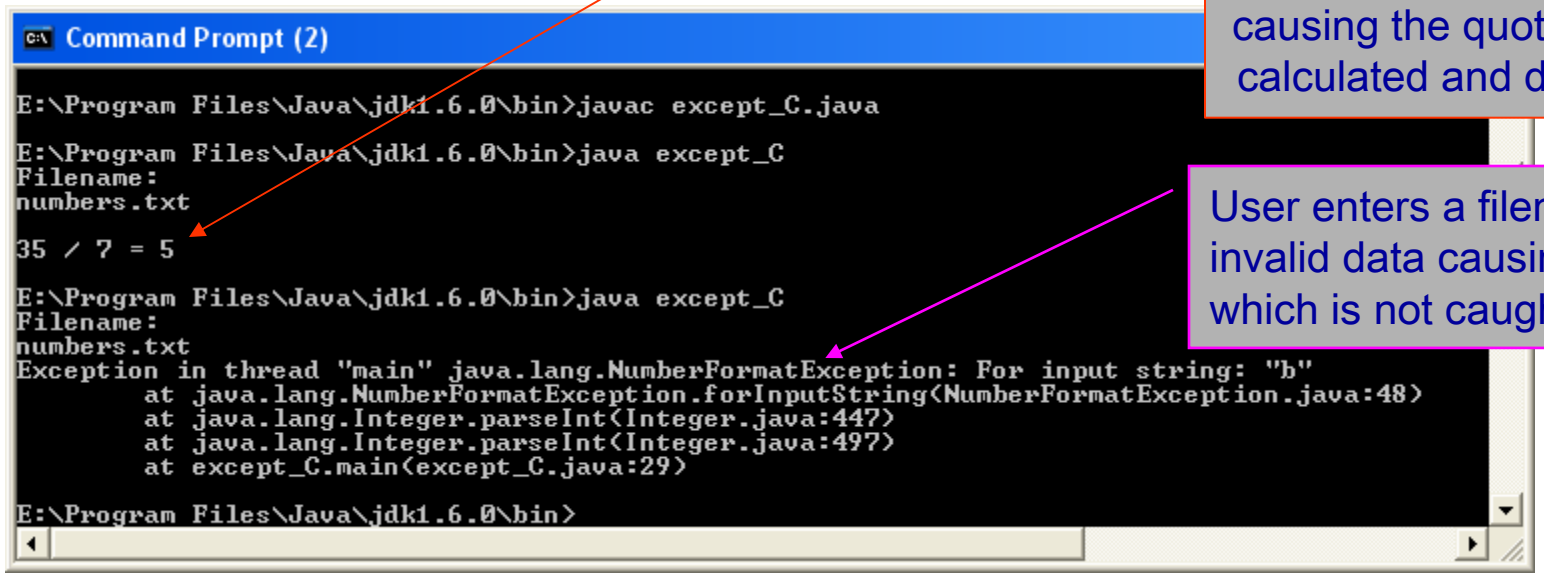
Program except_C.java

```
import java.io.*;

public class except_C {
   public static void main (String[] args){
     //get filename
     BufferedReader stdin = new BufferedReader(
               new InputStreamReader(System.in));
     System.out.println("Filename: ");
     String s = null;
     try {
        s = stdin.readLine();
     }
     catch (IOException e){
        System.err.println("Cannot read input");
        System.exit(0);
     }
     //set up file stream for processing
          BufferedReader filein = null;
     try {
          filein = new BufferedReader(new FileReader(s));
     }
```


```
      catch (FileNotFoundException e) {
            System.err.println(s + ": cannot be opened for
                        reading");
            System.exit(0);
      }

      //extract values and perform calculation
      try {
         int numerator = Integer.parseInt(filein.readLine());
         int denominator = Integer.parseInt(filein.readLine());
         int quotient = numerator / denominator;
         System.out.println();
         System.out.println(numerator + " / " + denominator + " = "
            + quotient);
      }
      catch (IOException e) {
         System.err.println(s + ": unable to read values");
         System.exit(0);
      }
      return;
   }
}
```


Output from except_C.java

- Suppose that program except_C is executed and the user specifies the file to be the file named numbers.txt containing the values 35 and 7 on successive lines. However, if that file contains the values 35 and b, then problems will occur as shown below.

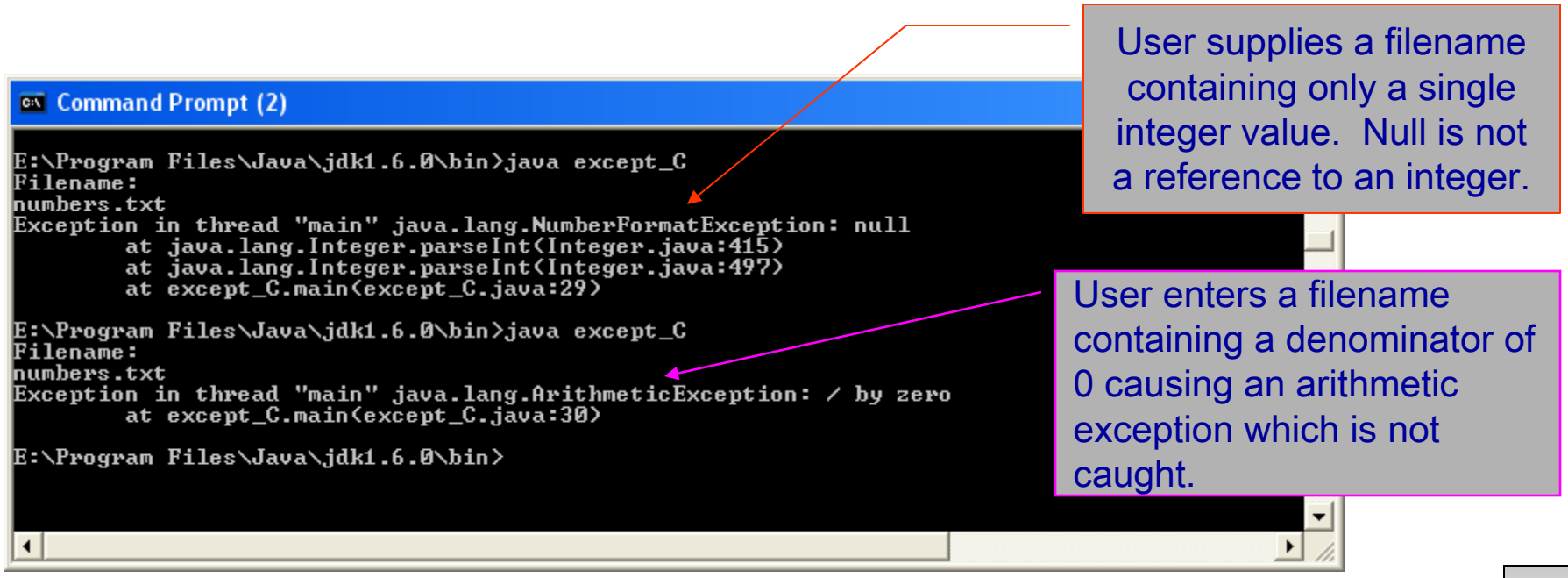


User supplies a filename containing valid data causing the quotient to be calculated and displayed.

User enters a filename containing invalid data causing an exception which is not caught.

Output from except_C.java (cont.)

- What would you expect to happen if program except_C is executed and the user specifies a file named numbers.txt containing only the value 35? Similarly, what would you expect to happen if the file contained the values 35 and 0? What happens is shown below.



Exception Handling in Java (cont.)

- The last three types of exceptions shown from executing program `except_C`, namely the two `NumberFormatException` and the one `ArithmeticException`, are examples of **runtime exceptions**.

- The superclass for all runtime exceptions is `java.lang.RunTimeException.` Because runtime exceptions can occur throughout a program and because of the cost of implementing handlers for them typically exceeds the expected benefit, Java makes it optional to catch them or to specify that it throws them.

Exception Handling in Java (cont.)

- Because runtime exceptions need not be caught, they are also known as **unchecked exceptions**.
- All other exceptions are known as checked exceptions. The checked exceptions that a method may generate **must** either be caught by one of its exception handlers or listed in the `throws` expression of the method.
- Program except_D (next three pages) adds exception handlers for these three additional exceptions.

Program except_D.java

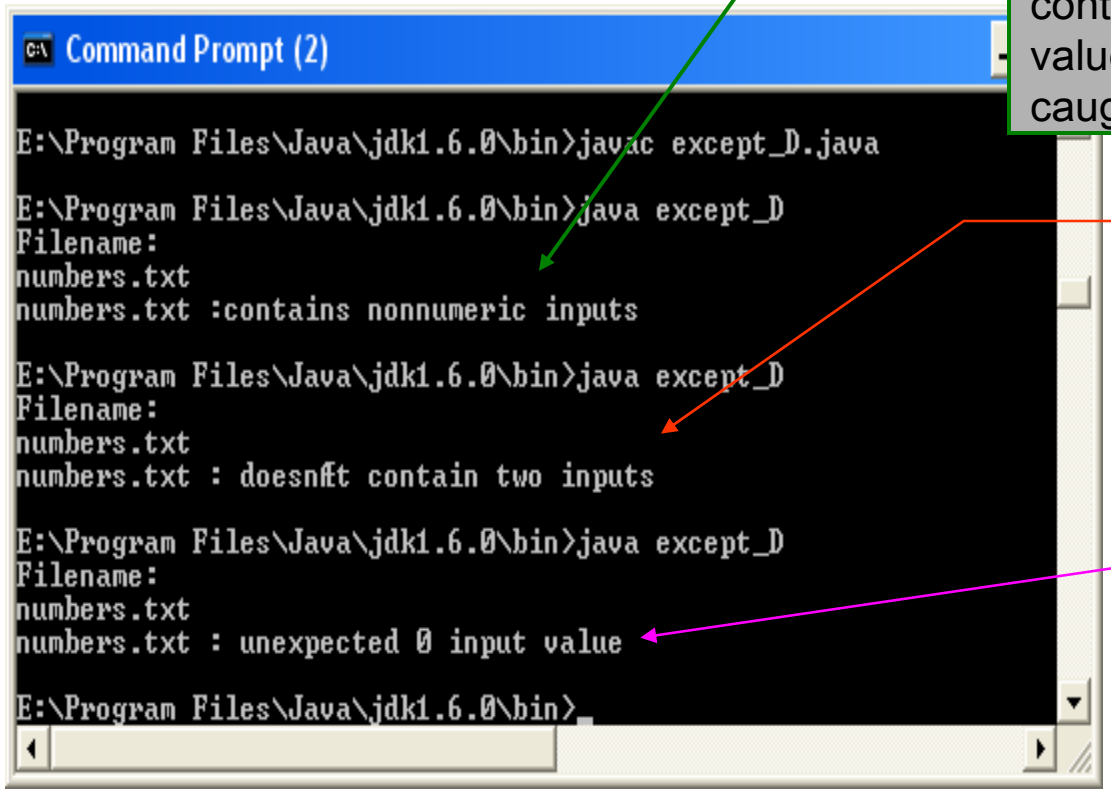
```
import java.io.*;

public class except_D {
   public static void main (String[] args){
     //get filename
     BufferedReader stdin = new BufferedReader(
               new InputStreamReader(System.in));
     System.out.println("Filename: ");
     String s = null;
     try {
        s = stdin.readLine();
     }
     catch (IOException e){
        System.err.println("Cannot read input");
        System.exit(0);
     }
     //set up file stream for processing
          BufferedReader filein = null;
     try {
          filein = new BufferedReader(new FileReader(s));
     }
```


```
 catch (FileNotFoundException e) {
      System.err.println(s + ": cannot be opened for reading");
      System.exit(0);
}
//extract values and perform calculation
try {
   int numerator = Integer.parseInt(filein.readLine());
   int denominator = Integer.parseInt(filein.readLine());
   int quotient = numerator / denominator;
   System.out.println();
   System.out.println(numerator + " / " + denominator + " = "
      + quotient);
}
catch (IOException e) {
   System.err.println(s + ": unable to read values");
   System.exit(0);
}
catch (NumberFormatException e) {
   if (e.getMessage().equals("null")) {
      System.err.println(s + " : doesn't contain two inputs");
   }
   else {
      System.err.println(s + " :contains nonnumeric inputs");
   }
   System.exit(0);
}
```


```
        catch (ArithmeticException e) {
           System.err.println(s + " : unexpected 0 input value");
           System.exit(0);
        }
        return;
    }
}
```


Output from except_D.java

- Using the same input files as before, the output from except_D is shown below:



User enters a filename containing a nonnumeric value. This exception is now caught.

User supplies a filename containing only a single integer value. This exception is now caught.

User enters a filename containing a denominator of 0 causing an arithmetic exception which is now caught.

Exception Handling in Java (cont.)

- For the particular problem that we have been using as an example, the handling of runtime exceptions makes sense because well-crafted programs ensure the validity of their input.
- Program `except_D` ensures that the input is valid by adding two more catch blocks to the try block for computing the quotient.
  - To handle the `NumberFormatException` the handler tests the return value of `Exception` instance method `getMessage()`. This method returns a `String` message indicating why the exception was thrown. Class method `Integer.parseInt()` throws an exception with message "null" only if its actual parameter has value `null`. Thus, the `e.getMessage().equals ("null")` test is sufficient for determining why the exception was generated.

Why Handle Exceptions

- Although Java's exception handling mechanism sometimes appears excessive, the mechanism is necessary if programs are to follow the object-oriented paradigm.
- For example, suppose an error occurred within a method `f()` that was invoked by a method `g()` that was in turn invoked by a method `h()`. Suppose further that to correct the error, method `h()` that initiated this invocation sequence must regain the flow of control. Without the exception-handling mechanism there would be no way to unwind the method invocations to enable corrective action to take place at the true problem source.

Finally – the last word in exception handling

- Although it is not strictly necessary, Java provides a syntax for an exception handler block that is always executed after the `try` block or `catch` handler have completed their tasks. This special handler is introduced through the keyword **`finally`**.
- The following program uses a finally handler. This program displays to standard output the contents of the files whose names are given to the program as command-line parameters.
  - The program Display.java mimics the operation of the Windows command `type` and the Unix/Linux command `cat`.

Program Display.java

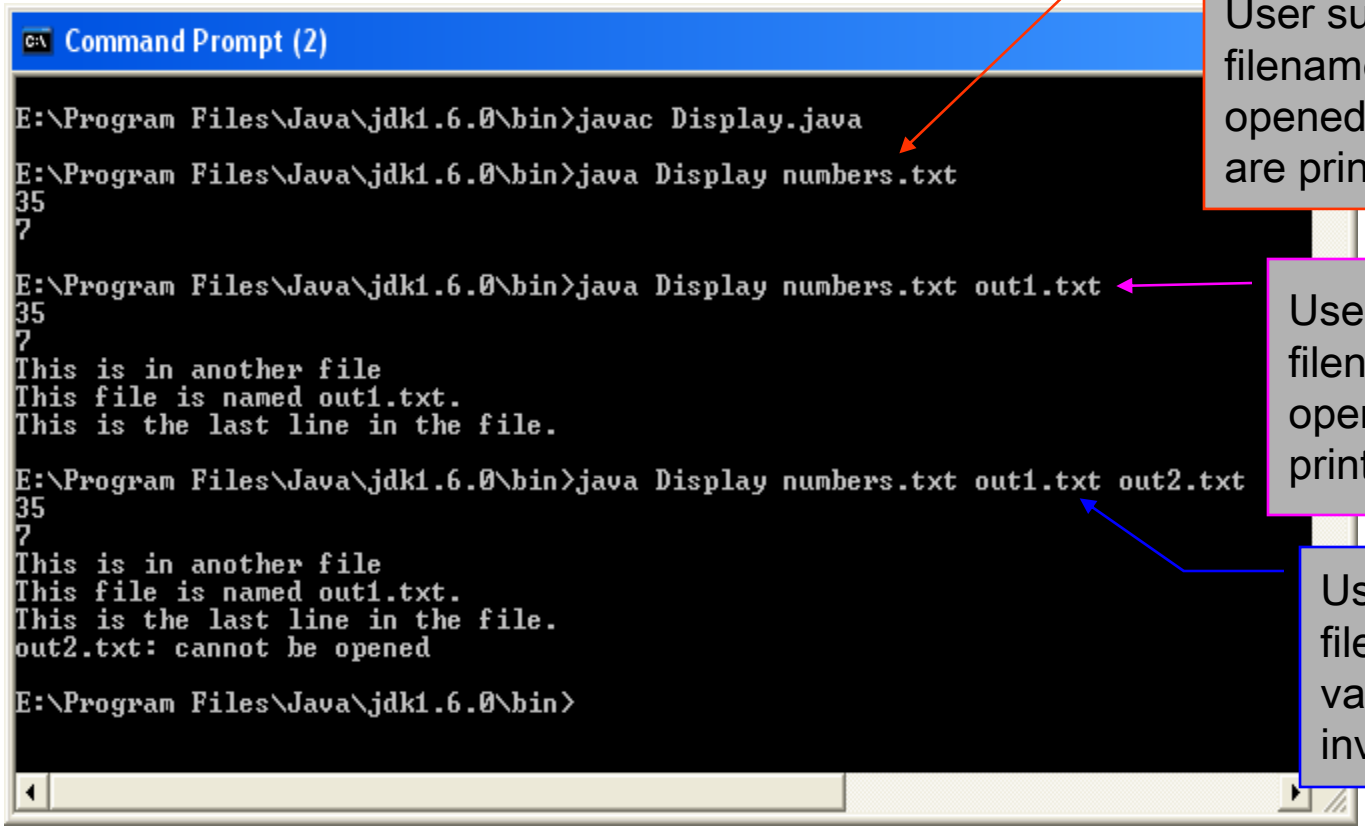
```
//This program mimics OS commands like type and cat
import java.io.*;
public class Display {
   public static void main (String[] args){
     //each command line parameter is treated as a filename
     //whose contents will be displayed to the standard output.
     for (int i = 0; i < args.length; ++i) {
        //open input stream reader associated with i-th file parameter
        try {
           BufferedReader filein = new BufferedReader(
                  new FileReader(args[i]));
           //args[i] is a readable filename
           try {
              String s = filein.readLine();
              while (s != null) {
                  System.out.println(s);
                  s = filein.readLine();
              }
           }
           catch (IOException e){
              System.err.println(args[i] + ": processing error");
           }
```


```
            finally {
               try {
                  filein.close();
               }
               catch (IOException e) {
                 System.err.println(args[i] + ": system error");
               }
            }
         }
         catch (FileNotFoundException e){
             //args[i] is not a valid filename
             System.err.println(args[i] + ": cannot be opened");
         }
      }
   }
}
```

The invocation `close()` generates an `IOException` if there is a file-system problem, so the `close()` is embedded within its own `try-catch` construct.

Output from `Display.java`

- `Display.java` acts like OS commands `cat` and `type`. Its command line arguments are opened and printed.



User supplies a valid filename. The file is opened and its contents are printed.

User enters two valid filenames. Both files are opened and their contents printed.

User enters three filenames. First two are valid and the third one is invalid (it does not exist).

Creating and Throwing Exceptions

- The keyword `throw` has two uses in Java. So far we have used it to head a `try` block. Its other use is in signaling an exception.
- A statement of the form: **`throw exception`**; is an exception throwing statement, where `exception` specifies the necessary exception information.
- In the next example, we'll return to the banking example that we used earlier in the term to create a `BankAccount` class which will throw a `NegativeAmountException` if there is an attempt to make a balance negative, deposit a negative amount, or withdraw a negative amount.

Creating and Throwing Exceptions (cont.)

- Since `NegativeAmountException` is not a built-in exception in Java, we will need to create the class `NegativeAmountException`.
- Class `BankAccount` supports two constructors: a default constructor to create a new bank account with an empty balance and an overloaded constructor which creates a new bank account with a positive balance.
- The overloaded constructor will throw a `NegativeAccountBalance` if an attempt is made to create an account with a negative initial balance.

Creating an Exception Class

- Class `NegativeAmountException` is a specialized exception class for indicating abnormal bank account manipulation. The behavior that we want from this exception is just the normal exception behavior, (e.g., using inherited method `getMessage()` to query an exception regarding its message).

```
public NegativeAmountException(String s)
```

- The definition of the `NegativeAmountException` constructor is straightforward. It simply invokes the constructor of its superclass `Exception` with `s` as the message for the new exception.

Creating an Exception Class

```
//represents an abnormal bank account event
public class NegativeAmountException extends Exception{
        //NegativeAmountException():creates exception with message s
   public NegativeAmountException(String s){
      super(s);
   }
}
```


BankAccount.java

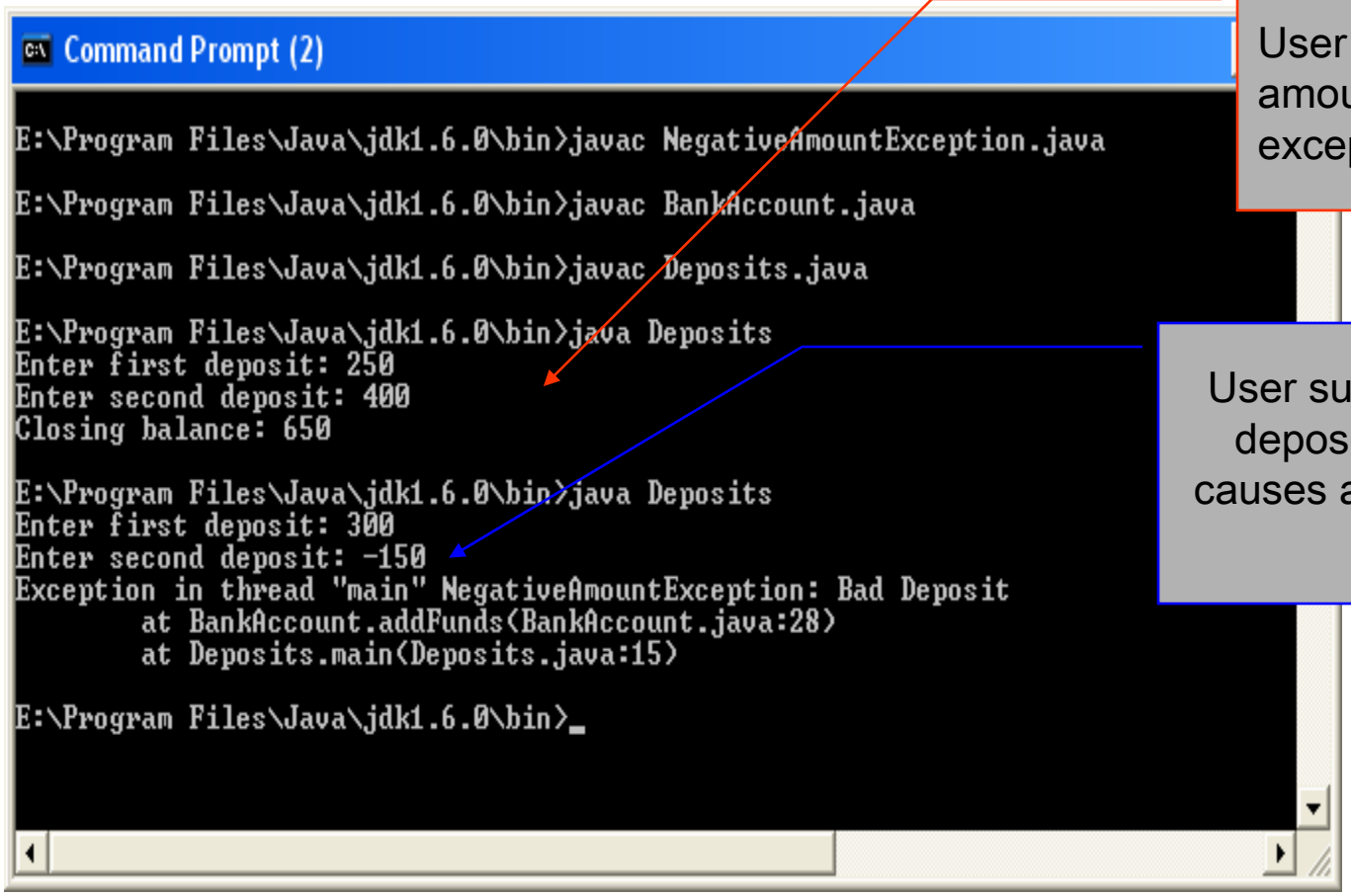
```
//BankAccount represents a bank account balance
public class BankAccount {
  //instance variable
  int balance;
  //BankAccount(): default constructor
  public BankAccount() {
    balance = 0;
  }
  //BankAccount(): overloaded specific constructor
  public BankAccount(int n) throws NegativeAmountException {
    if (n >= 0) {
       balance = n;
    }
    else {
       throw new NegativeAmountException("Bad Balance");
    }
  }
```


```
    //getBalance(): return the current balance
    public int getBalance() {
       return balance;
    }
    //addFunds(): deposit amount n
    public void addFunds(int n) throws NegativeAmountException {
       if (n >=0) {
           balance = += n;
       }
       else {
           throw new NegativeAmountException("Bad deposit");
       }
    }

    //removeFunds(): withdraw amount n
    public void removeFunds(int n) throws NegativeAmountException {
      if (n < 0) {
         throw new NegativeAmountException("Bad withdrawal");
      }
      else if (balance < n) {
          throw new NegativeAmountException("Bad balance");
      }
      else {
         balance -= n;
      }
    }
}
```


Deposits.java

```
//Demonstrate the use of BankAccount and NegativeAmountException
import java.io.*;

public class Deposits {
  //main(): application entry point
  public static void main (String[] args)throws IOException,
                                                 NegativeAmountException {
     BufferedReader stdin = new BufferedReader(new InputStreamReader
                                (System.in));
      BankAccount savings = new BankAccount();
      System.out.println("Enter first deposit");
      int deposit = Integer.parseInt(stdin.readLine());
      savings.addFunds(deposit);
      System.out.println("Enter second deposit");
      deposit = Integer.parseInt(stdin.readLine());
      savings.addFunds(deposit);
      System.out.println("Closing balance: " + savings.getBalance());
   }
}
```


Output from `Deposits.java`



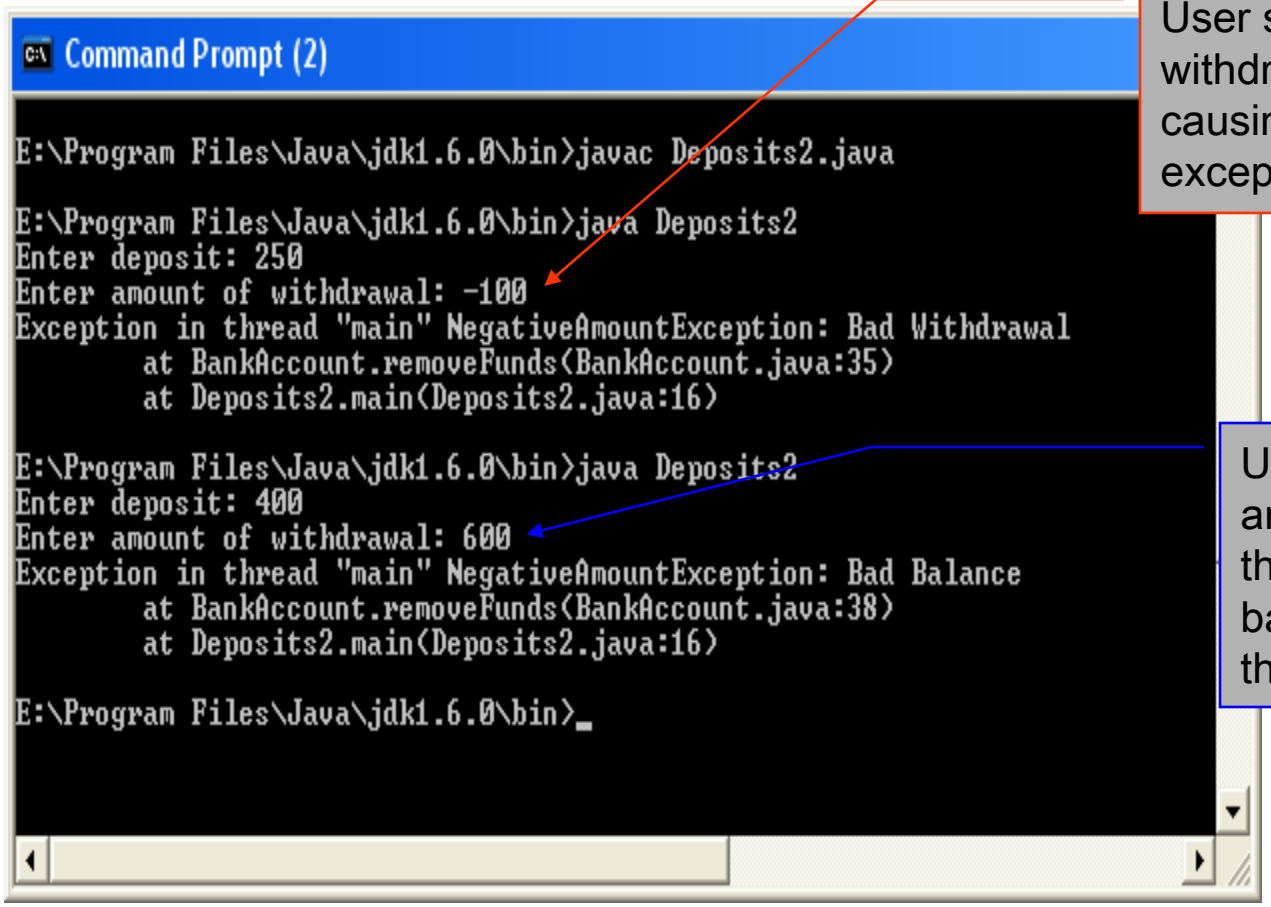
User supplies valid amounts for deposits. No exceptions are thrown.

User supplies a negative deposit amount which causes an exception to be thrown

Modified_Deposits.java

```
//Demostrate the use of BankAccount and NegativeAmountException
//Illustrates two exceptions caused by withdrawals
import java.io.*;

public class Deposits2 {
  //main(): application entry point
  public static void main (String[] args)throws IOException,
                                                 NegativeAmountException {
     BufferedReader stdin = new BufferedReader(new InputStreamReader
                                (System.in));
      BankAccount savings = new BankAccount();
      System.out.printl("Enter deposit: ");
      int deposit = Integer.parseInt(stdin.readLine());
      savings.addFunds(deposit);
      System.out.printl("Enter amount of withdrawal: ");
      int withdrawal = Integer.parseInt(stdin.readLine());
      savings.removeFunds(withdrawal);
      System.out.println("Closing balance: " + savings.getBalance());
   }
}
```


Output from `Modified_Deposits.java`



User supplies a negative withdrawal amount causing a bad withdrawal exception.

User supplies a withdrawal amount which is larger than the balance causing a bad balance exception to be thrown.