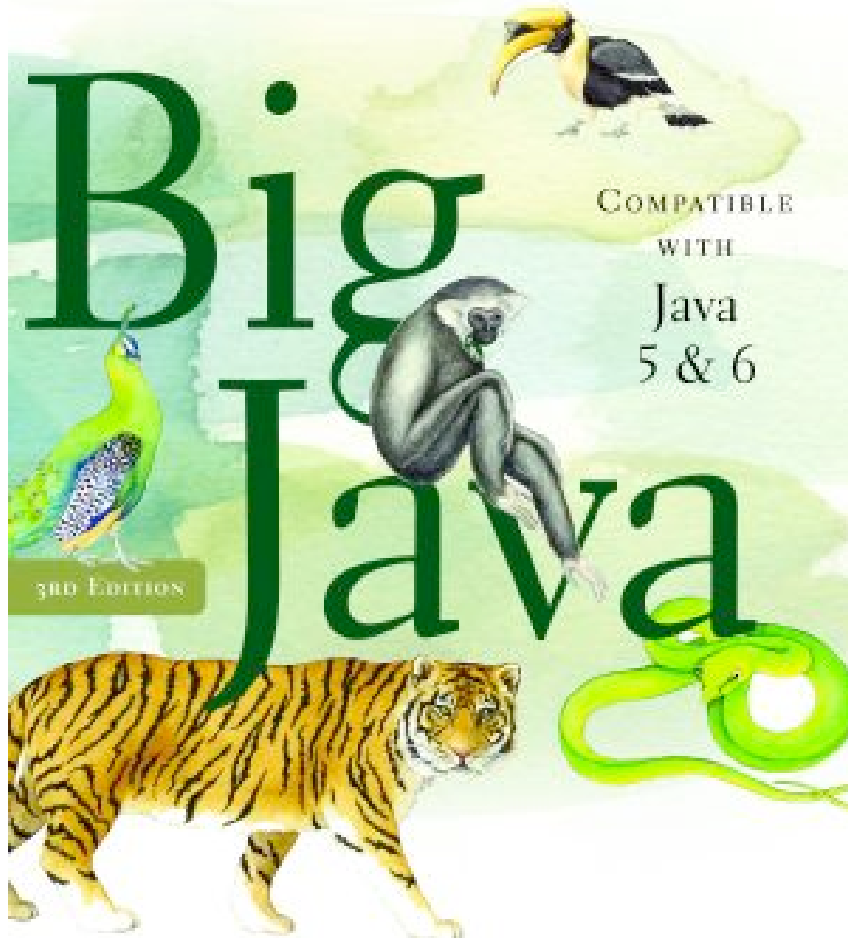


CAY HORSTMANN



## Chapter Eleven: Input/Output and Exception Handling

*Big Java* by Cay Horstmann  
Copyright © 2008 by John Wiley & Sons. All rights reserved.

## Chapter Goals

---

- To be able to read and write text files
- 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

## Reading Text Files

---

- Simplest way to read text: use `Scanner` class
- To read from a disk file, construct a `FileReader`
- Then, use the `FileReader` to construct a `Scanner` object

```
FileReader reader = new FileReader("input.txt");
Scanner in = new Scanner(reader);
```
- Use the `Scanner` methods to read data from file  
`next`, `nextLine`, `nextInt`, and `nextDouble`

## Writing Text Files

---

- To write to a file, construct a `PrintWriter` object  
`PrintWriter out = new PrintWriter("output.txt");`
- If file already exists, it is emptied before the new data are written into it
- If file doesn't exist, an empty file is created
- Use `print` and `println` to write into a `PrintWriter`:  
`out.println(29.95);`  
`out.println(new Rectangle(5, 10, 15, 25));`  
`out.println("Hello, World!");`
- You must close a file when you are done processing it:  
`out.close();`

Otherwise, not all of the output may be written to the disk file

## FileNotFoundException

---

- When the input or output file doesn't exist, a `FileNotFoundException` can occur
- To handle the exception, label the main method like this:  
`public static void main(String[] args) throws  
    FileNotFoundException`

## A Sample Program

---

- Reads all lines of a file and sends them to the output file, preceded by line numbers
- Sample input file:  
Mary had a little lamb  
Whose fleece was white as snow.  
And everywhere that Mary went,  
The lamb was sure to go!
- Program produces the output file:  
/\* 1 \*/ Mary had a little lamb  
/\* 2 \*/ Whose fleece was white as snow.  
/\* 3 \*/ And everywhere that Mary went,  
/\* 4 \*/ The lamb was sure to go!
- Program can be used for numbering Java source files

## ch11/fileio/LineNumberer.java

```
01: import java.io.FileReader;
02: import java.io.FileNotFoundException;
03: import java.io.PrintWriter;
04: import java.util.Scanner;
05:
06: public class LineNumberer
07: {
08:     public static void main(String[] args)
09:         throws FileNotFoundException
10:     {
11:         Scanner console = new Scanner(System.in);
12:         System.out.print("Input file: ");
13:         String inputFileName = console.next();
14:         System.out.print("Output file: ");
15:         String outputFileName = console.next();
16:
17:         FileReader reader = new FileReader(inputFileName);
18:         Scanner in = new Scanner(reader);
19:         PrintWriter out = new PrintWriter(outputFileName);
20:         int lineNumber = 1;
```

**Continued**

*Big Java* by Cay Horstmann  
Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch11/fileio/LineNumberer.java (cont.)

```
21:
22:     while (in.hasNextLine())
23:     {
24:         String line = in.nextLine();
25:         out.println("/ * " + lineNumber + " */ " + line);
26:         lineNumber++;
27:     }
28:
29:     out.close();
30: }
31: }
```



## Self Check 11.1

---

What happens when you supply the same name for the input and output files to the `LineNumberer` program?

**Answer:** When the `PrintWriter` object is created, the output file is emptied. Sadly, that is the same file as the input file. The input file is now empty and the `while` loop exits immediately.

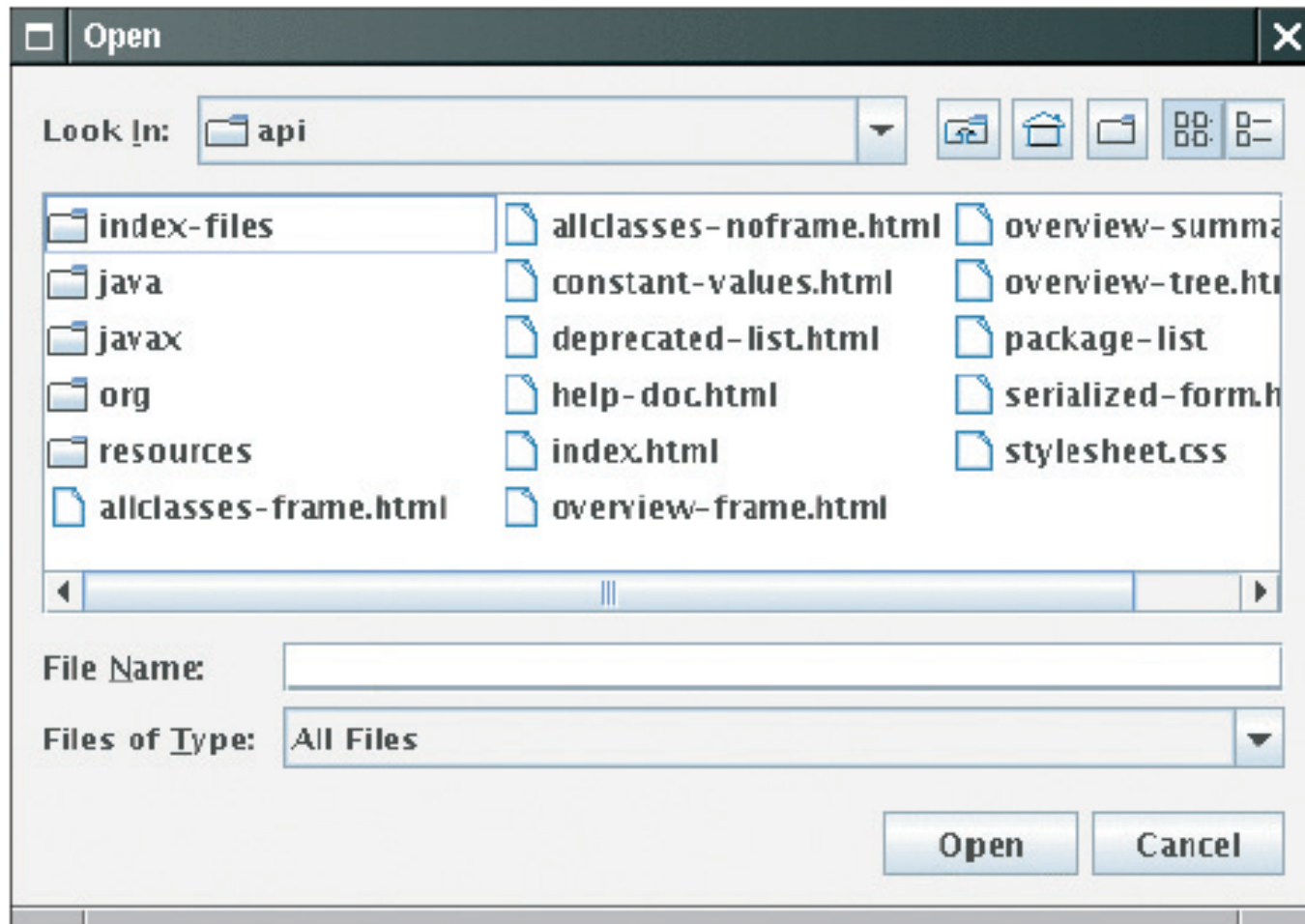
## Self Check 11.2

---

What happens when you supply the name of a nonexistent input file to the `LineNumberer` program?

**Answer:** The program catches a `FileNotFoundException`, prints an error message, and terminates.

## File Dialog Boxes



A JFileChooser Dialog Box

***Continued***

## File Dialog Boxes (cont.)

---

```
JFileChooser chooser = new JFileChooser();
FileReader in = null;
if (chooser.showOpenDialog(null) ==
    JFileChooser.APPROVE_OPTION)
{
    File selectedFile = chooser.getSelectedFile();
    reader = new FileReader(selectedFile);
    . . .
}
```

## Throwing Exceptions

- Throw an exception object to signal an exceptional condition
- **Example:** `IllegalArgumentException`: illegal parameter value

```
IllegalArgumentException exception
    = new IllegalArgumentException("Amount exceeds
        balance");
throw exception;
```
- **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

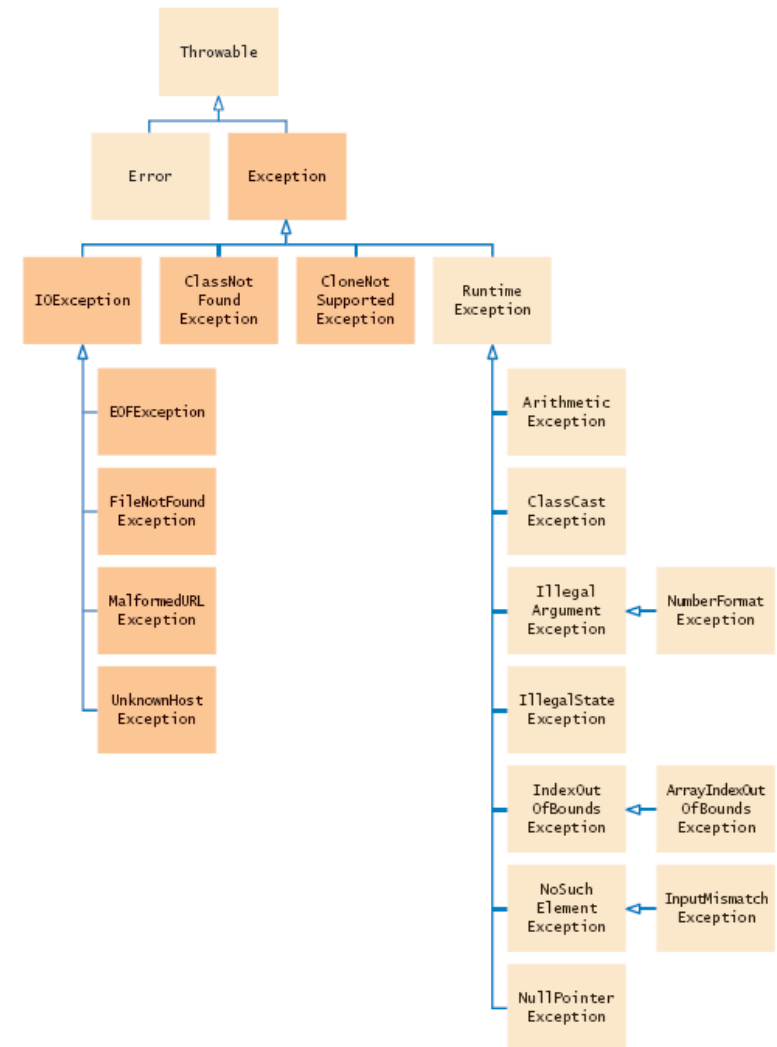


Figure 1 The Hierarchy of Exception Classes

## Syntax 11.1 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 11.3

---

How should you modify the `deposit` method to ensure that the balance is never negative?

**Answer:** Throw an exception if the amount being deposited is less than zero.

## Self Check 11.4

---

Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`. What is the value of `balance` afterwards?

**Answer:** 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*
    - o The compiler checks that you don't ignore them
    - o Due to external circumstances that the programmer cannot prevent
    - o Majority occur when dealing with input and output
    - o For example, `IOException`
  - *Unchecked:*
    - o Extend the class `RuntimeException` or `Error`
    - o They are the programmer's fault
    - o Examples of runtime exceptions:  
`NumberFormatException`  
`IllegalArgumentException`  
`NullPointerException`
    - o 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
- 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:

1. *Handle the exception*
2. *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);  
    . . .  
}
```

- For multiple exceptions:

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

***Continued***

## Checked and Unchecked Exceptions (cont.)

---

- 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 11.2 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 11.5

---

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?

**Answer:** The specification throws `IOException` is sufficient because `FileNotFoundException` is a subclass of `IOException`.



## Self Check 11.6

---

Why is a `NullPointerException` not a checked exception?

**Answer:** 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 (cont.)

- 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
- `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 11.3 General Try Block

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

***Continued***

## Syntax 11.3 General Try Block (cont.)

### 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();
}
```

***Continued***

## Syntax 11.3 General Try Block (cont.)

---

### 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 11.7

---

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.

**Answer:** 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.



## Self Check 11.8

---

Is there a difference between catching checked and unchecked exceptions?

**Answer:** No – you catch both exception types in the same way, as you can see from the code example on page 508. 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
```
- **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 11.4 The `finally` Clause

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

***Continued***

## Syntax 11.4 The `finally` Clause (cont.)

### 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 11.9

---

Why was the `out` variable declared outside the `try` block?

**Answer:** 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.

## Self Check 11.10

---

Suppose the file with the given name does not exist. Trace the flow of execution of the code segment in this section.

**Answer:** 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 Exception 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
- 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 Exception Types

---

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

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

## Self Check 11.11

---

What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

**Answer:** To pass the exception message string to the `RuntimeException` superclass.

## Self Check 11.12

---

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?

**Answer:** `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 Example

---

- 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 Example

---

- 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*
- 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 Example (cont.)

---

- 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*

## ch11/data/DataAnalyzer.java

```
01: import java.io.FileNotFoundException;
02: import java.io.IOException;
03: import java.util.Scanner;
04:
05: /**
06:     This program reads a file containing numbers and analyzes its contents.
07:     If the file doesn't exist or contains strings that are not numbers, an
08:     error message is displayed.
09: */
10: public class DataAnalyzer
11: {
12:     public static void main(String[] args)
13:     {
14:         Scanner in = new Scanner(System.in);
15:         DataSetReader reader = new DataSetReader();
16:
17:         boolean done = false;
18:         while (!done)
19:         {
20:             try
21:             {
22:                 System.out.println("Please enter the file name: ");
23:                 String filename = in.next();
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.



## ch11/data/DataAnalyzer.java (cont.)

```
24:
25:         double[] data = reader.readFile(filename);
26:         double sum = 0;
27:         for (double d : data) sum = sum + d;
28:         System.out.println("The sum is " + sum);
29:         done = true;
30:     }
31:     catch (FileNotFoundException exception)
32:     {
33:         System.out.println("File not found.");
34:     }
35:     catch (BadDataException exception)
36:     {
37:         System.out.println("Bad data: " + exception.getMessage());
38:     }
39:     catch (IOException exception)
40:     {
41:         exception.printStackTrace();
42:     }
43: }
44: }
45: }
```

## The `readFile` method of the `DataSetReader` class

---

- Constructs `Scanner` object
- Calls `readData` method
- Completely unconcerned with any exceptions
- If there is a problem with input file, it simply passes the exception to caller

***Continued***

## The `readFile` method of the `DataSetReader` class (cont.)

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

## The `readData` 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");
}
```

***Continued***

## The `readData` method of the `DataSetReader` class (cont.)

---

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

## The readValue 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();
}
```

## Animation 11.1 –

```
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         }
32         return data;
33     }
34
35     /**
36      Reads all data.
37      @param in the scanner that scans the data
38      */
```

This animation walks through an exception handling scenario with the **DataAnalyzer** class from Chapter 11. You will learn about throwing exceptions, catching exceptions, and the **finally** clause.



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



## ch11/data/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: {
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);
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch11/data/DataSetReader.java (cont.)

```
23:         try
24:         {
25:             Scanner in = new Scanner(reader);
26:             readData(in);
27:         }
28:         finally
29:         {
30:             reader.close();
31:         }
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];
```

**Continued**

*Big Java* by Cay Horstmann

Copyright © 2008 by John Wiley & Sons. All rights reserved.

## ch11/data/DataSetReader.java (cont.)

```
45:
46:     for (int i = 0; i < numberOfValues; i++)
47:         readValue(in, i);
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: {
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 11.13

---

Why doesn't the `DataSetReader.readFile` method catch any exceptions?

**Answer:** 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.

## Self Check 11.14

---

Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

**Answer:** `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.