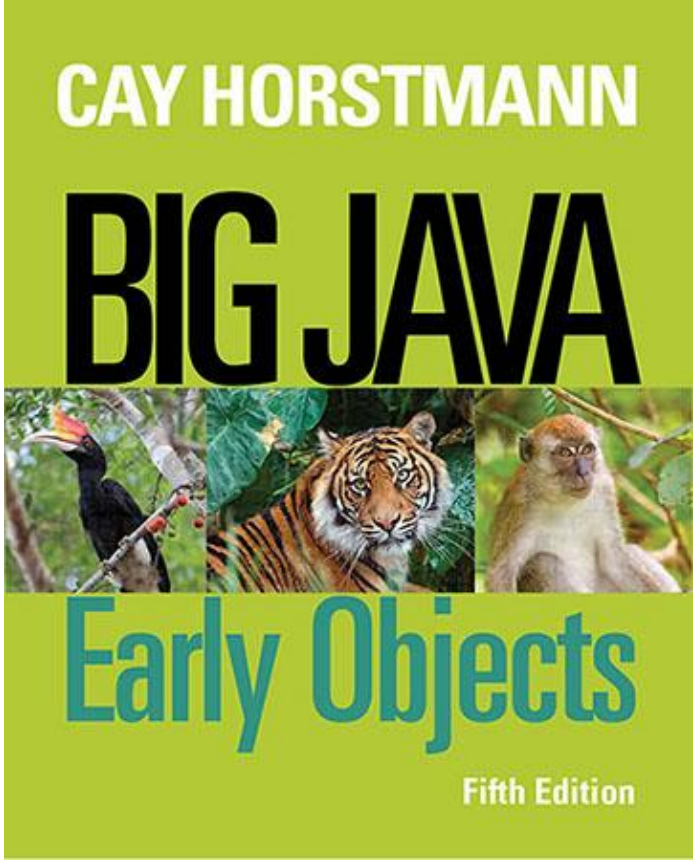


Chapter 11 – Input/Output and Exception Handling

Exception Handling - Throwing Exceptions

- Exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.
- When you detect an error condition, throw an exception object to signal an exceptional condition
- If someone tries to withdraw too much money from a bank account
  - Throw an `IllegalArgumentException`

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

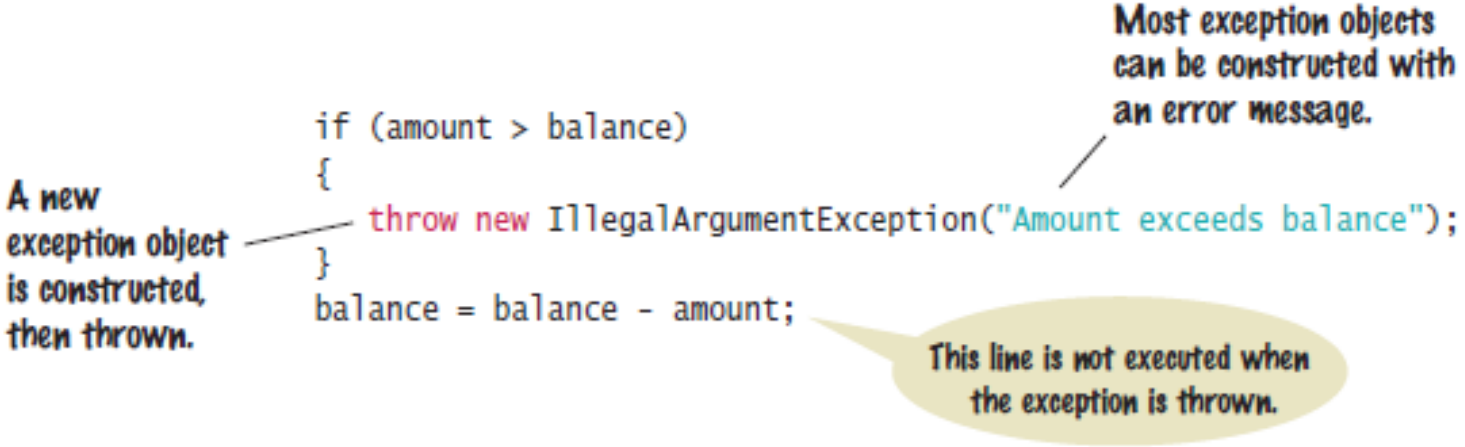
Exception Handling - Throwing Exceptions

- When an exception is thrown, method terminates immediately
  - Execution continues with an exception handler
- When you throw an exception, the normal control flow is terminated. This is similar to a circuit breaker that cuts off the flow of electricity in a dangerous situation.

© Lisa F. Young/iStockphoto.

Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;



Hierarchy of Exception Classes

Figure 2 A Part of the Hierarchy of Exception Classes

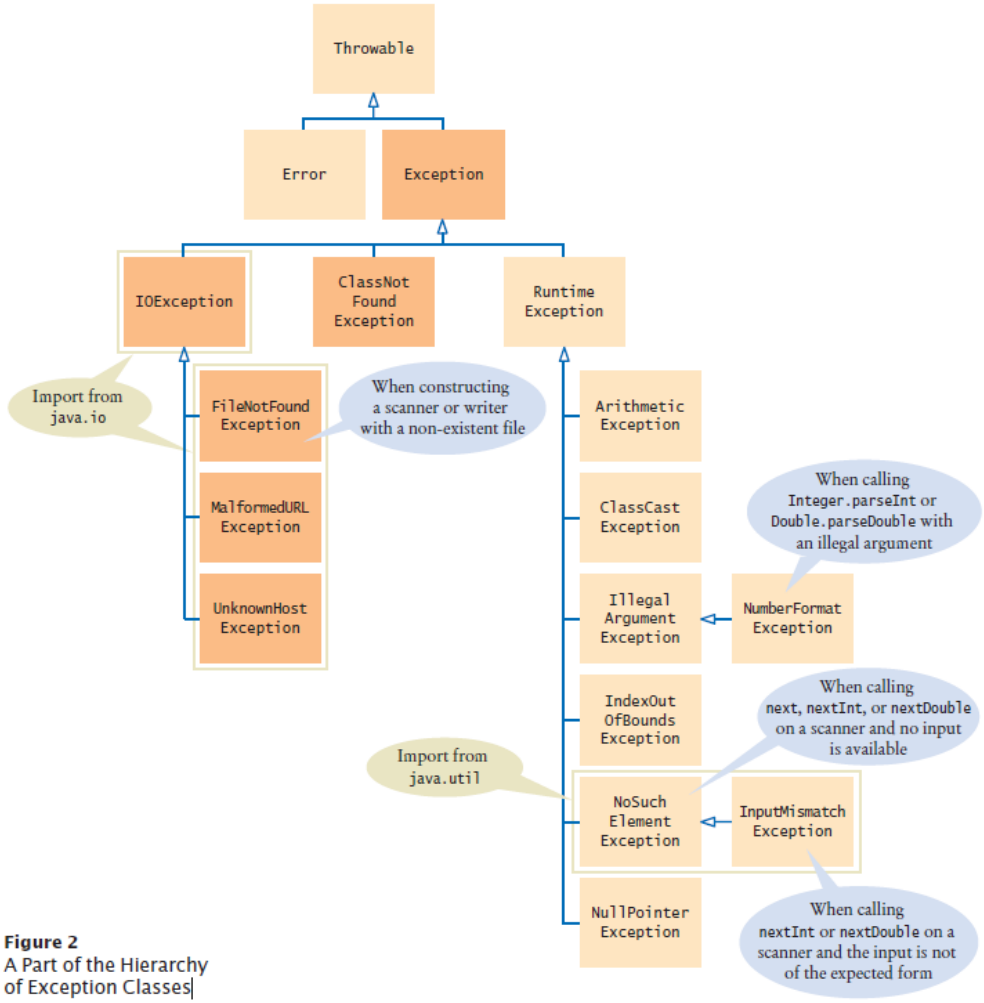


Figure 2
A Part of the Hierarchy of Exception Classes

Catching Exceptions

- Every exception should be handled somewhere in your program
- Place the statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause.

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println(exception.getMessage());
}
```

Catching Exceptions

- Three exceptions may be thrown in the `try` block:
  - The `Scanner` constructor can throw a `FileNotFoundException`.
  - `Scanner.next` can throw a `NoSuchElementException`.
  - `Integer.parseInt` can throw a `NumberFormatException`.
- If any of these exceptions is actually thrown, then the rest of the instructions in the `try` block are skipped.

Catching Exceptions

- What happens when each exception is thrown:
- If a `FileNotFoundException` is thrown,
  - then the `catch` clause for the `IOException` is executed because `FileNotFoundException` is a descendant of `IOException`.
  - If you want to show the user a different message for a `FileNotFoundException`, you must place the `catch` clause before the clause for an `IOException`
- If a `NumberFormatException` occurs,
  - then the second `catch` clause is executed.
- A `NoSuchElementException` is not caught by any of the `catch` clauses.
  - The exception remains thrown until it is caught by another `try` block.

Syntax 11.2 Catching Exceptions

Syntax

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

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage());
}
```

This constructor can throw a FileNotFoundException.

When an IOException is thrown, execution resumes here.

This is the exception that was thrown.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

A FileNotFoundException is a special case of an IOException.

Catching Exceptions

- Each `catch` clause contains a handler.
- Our example just informed the user of a problem.
- Often better to give the user another chance.
- When you throw an exception, you can provide your own message string.
- For example, when you call

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

  the message of the exception is the string provided in the constructor.
- You should only catch those exceptions that you can handle.

© Andraz Cerar/iStockphoto.

Checked Exceptions

- Exceptions fall into three categories
- Internal errors are reported by descendants of the type `Error`.
  - Example: `OutOfMemoryError`
- Descendants of `RuntimeException`,
  - Example: `IndexOutOfBoundsException` or `IllegalArgumentException`
  - Indicate errors in your code.
  - They are called unchecked exceptions.
- All other exceptions are checked exceptions.
  - Indicate that something has gone wrong for some external reason beyond your control
  - Example: `IOException`

Checked Exceptions

- Checked exceptions are due to external circumstances that the programmer cannot prevent.
  - The compiler checks that your program handles these exceptions.
- The unchecked exceptions are your fault.
  - The compiler does not check whether you handle an unchecked exception.

Checked Exceptions - throws

- You can handle the checked exception in the same method that throws it

```
try
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile); // ThrowsFileNotFoundException
   . . .
}
catch (FileNotFoundException exception) // Exception caught here
{
   . . .
}
```

Checked Exceptions - throws

- Often the current method cannot handle the exception. Tell the compiler you are aware of the exception
- You want the method to terminate if the exception occurs
- Add a throws clause to the method header

```
public void readData(String filename) throws FileNotFoundException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   . . .
}
```

Checked Exceptions - `throws`

- The `throws` clause signals to the caller of your method that it may encounter a `FileNotFoundException`.
  - The caller must decide
    - To handle the exception
    - Or declare the exception may be thrown
- Throw early, catch late
  - Throw an exception as soon as a problem is detected.
  - Catch it only when the problem can be handled
- Just as trucks with large or hazardous loads carry warning signs, the throws clause warns the caller that an exception may occur.

© tillsonburg/iStockphoto.

Syntax 11.3 throws Clause

Syntax *modifiers returnType methodName(parameterType parameterName, . . .)*
throws *ExceptionClass, ExceptionClass, . . .*

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

The `finally` Clause

- Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed - whether or not an exception is thrown.
- Use when you do some clean up
- Example - closing files

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

- Executes the `close` even if an exception is thrown.

The `finally` Clause

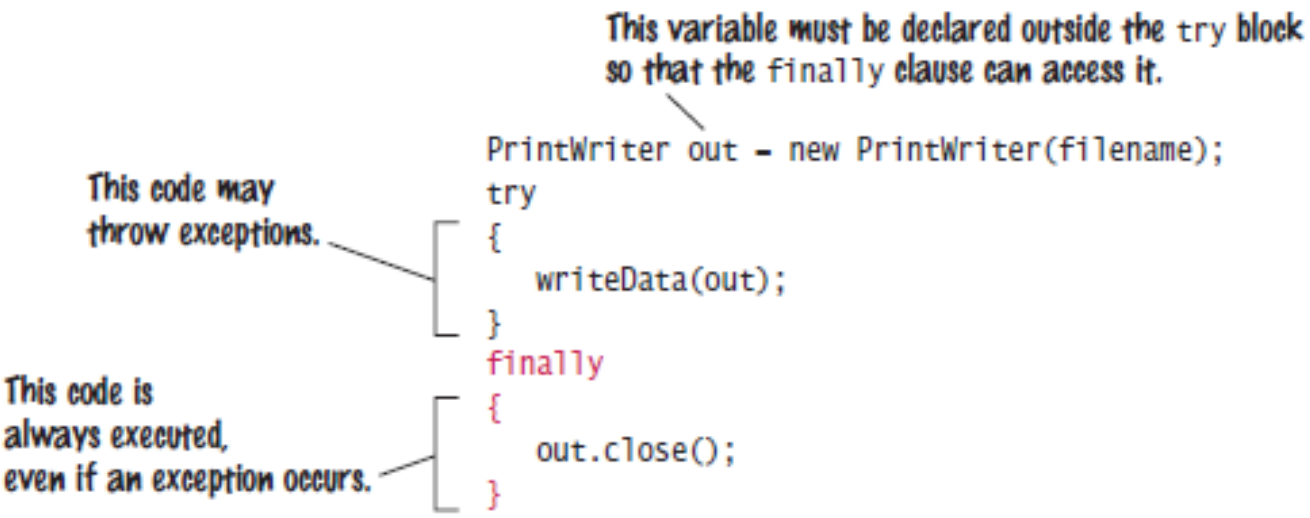
© archives/iStockphoto.

All visitors to a foreign country have to go through passport control, no matter what happened on their trip. Similarly, the code in a `finally` clause is always executed, even when an exception has occurred.

Syntax 11.4 `finally` Clause

Syntax

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



Designing Your Own Exception Types

- You can design your own exception types — subclasses of `Exception` or `RuntimeException`.
- Throw an `InsufficientFundsException` when the amount to withdraw an amount from a bank account exceeds the current balance.

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

- Make `InsufficientFundsException` an unchecked exception
  - Programmer could have avoided it by calling `getBalance` first
  - Extend `RuntimeException` or one of its subclasses

Designing Your Own Exception Types

- Supply two constructors for the class
  - A constructor with no arguments
  - A constructor that accepts a message string describing reason for exception

```
public class InsufficientFundsException extends RuntimeException
{
   public InsufficientFundsException() {}
   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

- When the exception is caught, its message string can be retrieved
  - Using the `getMessage` method of the `Throwable` class.

Self Check 11.16

Suppose `balance` is `100` and `amount` is `200`. What is the value of `balance` after these statements?

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Answer: It is still `100`. The last statement was not executed because the exception was thrown.

Self Check 11.17

When depositing an amount into a bank account, we don't have to worry about overdrafts—except when the amount is negative. Write a statement that throws an appropriate exception in that case.

Answer:

```
if (amount < 0)
{
   throw new IllegalArgumentException("Negative amount");
}
```

Self Check 11.18

Consider the method

```
public static void main(String[] args)
{
   try
   {
      Scanner in = new Scanner(new File("input.txt"));
      int value = in.nextInt();
      System.out.println(value);
   }
   catch (IOException exception)
   {
      System.out.println("Error opening file.");
   }
}
```

Suppose the file with the given file name exists and has no contents. Trace the flow of execution.

Continued

Self Check 11.18

Answer: The `Scanner` constructor succeeds because the file exists. The `nextInt` method throws a `NoSuchElementException`. This is not an `IOException`. Therefore, the error is not caught. Because there is no other handler, an error message is printed and the program terminates.

Self Check 11.19

Why is an `ArrayIndexOutOfBoundsException` not a checked exception?

Answer: Because programmers should simply check that their array index values are valid instead of trying to handle an `ArrayIndexOutOfBoundsException`.

Self Check 11.20

Is there a difference between catching checked and unchecked exceptions?

Answer: No. You can catch both exception types in the same way, as you can see in the code example on page 536.

Self Check 11.21

What is wrong with the following code, and how can you fix it?

```
public static void writeAll(String[] lines, String filename)
{
   PrintWriter out = new PrintWriter(filename);
   for (String line : lines)
   {
      out.println(line.toUpperCase());
   }
   out.close();
}
```

Answer: There are two mistakes. The `PrintWriter` constructor can throw a `FileNotFoundException`. You should supply a throws clause. And if one of the array elements is `null`, a `NullPointerException` is thrown. In that case, the `out.close()` statement is never executed. You should use a `try/finally` statement.

Self Check 11.22

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

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

Self Check 11.23

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: Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException` or `IllegalArgumentException`. Because the error is related to input, `IOException` would be a good choice.

Application: Handling Input Errors

- 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
```

Case Study: A Complete Example

- What can go wrong?
  - File might not exist
  - File might have data in wrong format
- Who can detect the faults?
  - `Scanner` 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 `Scanner` constructor
  - `BadDataException`, a custom checked exception class for reporting wrong data format

Case Study: A Complete Example

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

section_5/DataAnalyzer.java

```
1  import java.io.FileNotFoundException;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  /**
6     This program reads a file containing numbers and analyzes its contents.
7     If the file doesn&amp;apos;t exist or contains strings that are not numbers, an
8     error message is displayed.
9  */
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
```

Continued

section_5/DataAnalyzer.java

```
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();
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:

```
public double[] readFile(String filename) throws IOException
{
   File inFile = new File(filename);
   Scanner in = new Scanner(inFile);
   try
   {
      readData(in);
      return data;
   }
   finally { in.close(); }
}
```

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");
}
```

The `readData` 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 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();
}
```

Error Scenario

1. `DataAnalyzer.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 and closing the `Scanner` object
8. `DataAnalyzer.main` has handler for `BadDataException`
   - Handler prints a message
   - User is given another chance to enter file name

section_5/DataSetReader.java

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.Scanner;
4
5  /**
6     Reads a data set from a file. The file must have the format
7     numberOfValues
8     value1
9     value2
10    . . .
11 */
12 public class DataSetReader
13 {
14    private double[] data;
15
```

Continued

section_5/DataSetReader.java

```
16  /**
17     Reads a data set.
18     @param filename the name of the file holding the data
19     @return the data in the file
20  */
21  public double[] readFile(String filename) throws IOException
22  {
23     File inFile = new File(filename);
24     Scanner in = new Scanner(inFile);
25     try
26     {
27        readData(in);
28        return data;
29     }
30     finally
31     {
32        in.close();
33     }
34  }
35
```

Continued

section_5/DataSetReader.java

```
36    /**
37       Reads all data.
38       @param in the scanner that scans the data
39    */
40    private void readData(Scanner in) throws BadDataException
41    {
42       if (!in.hasNextInt())
43       {
44          throw new BadDataException("Length expected");
45       }
46       int numberOfValues = in.nextInt();
47       data = new double[numberOfValues];
48
49       for (int i = 0; i < numberOfValues; i++)
50       {
51          readValue(in, i);
52       }
53
54       if (in.hasNext())
55       {
56          throw new BadDataException("End of file expected");
57       }
58    }
59
```

Continued

section_5/DataSetReader.java

```
60    /**
61       Reads one data value.
62       @param in the scanner that scans the data
63       @param i the position of the value to read
64    */
65    private void readValue(Scanner in, int i) throws BadDataException
66    {
67       if (!in.hasNextDouble())
68       {
69          throw new BadDataException("Data value expected");
70       }
71       data[i] = in.nextDouble();
72    }
73 }
```

section_5/BadDataException.java

```
1  import java.io.IOException;
2
3  /**
4     This class reports bad input data.
5  */
6  public class BadDataException extends IOException
7  {
8     public BadDataException() {}
9     public BadDataException(String message)
10    {
11       super(message);
12    }
13 }
```

Self Check 11.24

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 11.25

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

Answer: `DataAnalyzer.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.

Self Check 11.26

If the `readValue` method had to throw a `NoSuchElementException` instead of a `BadDataException` when the next input isn't a floating-point number, how would the implementation change?

Answer: It could simply be

```
private void readValue(Scanner in, int i)
{
   data[i] = in.nextDouble();
}
```

The `nextDouble` method throws a `NoSuchElementException` or a `InputMismatchException` (which is a subclass of `NoSuchElementException`) when the next input isn't a floating-point number. That exception isn't a checked exception, so it need not be declared.

Self Check 11.27

Consider the `try/finally` statement in the `readFile` method. Why was the `in` variable declared outside the `try` block?

Answer: If it had been declared inside the `try` block, its scope would only have extended until the end of the `try` block, and it would not have been accessible in the `finally` clause.

Self Check 11.28

How can the program be simplified when you use the "automatic resource management" feature described in Special Topic 11.6?

Answer: The `try/finally` statement in the `readFile` method can be rewritten as

```
try (Scanner in = new Scanner(inFile))
{
   readData(in);
   return data;
}
```