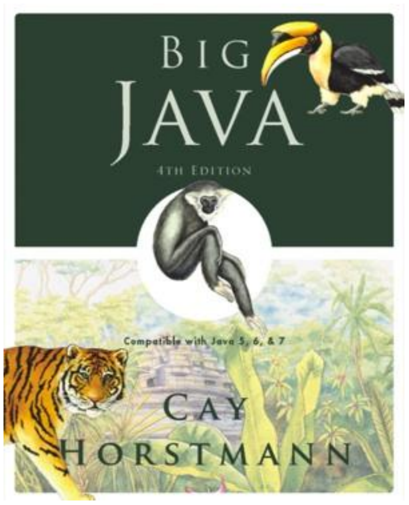


Chapter 11 – Input/Output and Exception Handling

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 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

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.

Reading Text Input: Reading Words

- The `next` method reads a word at a time:

```
while (in.hasNext())
{
   String input = in.next();
   System.out.println(input);
}
```

- With our sample input, the output is:

```
Mary
had
a
little
lamb
…
```

- A *word* is any sequence of characters that is not white space

Reading Text Input: Processing Lines

- Then use the `isDigit` and `isWhitespace` methods to find out where the name ends and the number starts. E.g. locate the first digit:

```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

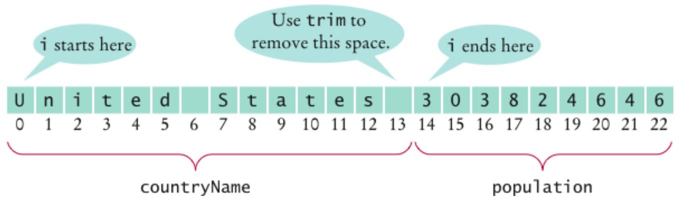
- Then extract the country name and population:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

Reading Text Input: Processing Lines

- Use the `trim` method to remove spaces at the end of the country name:

```
countryName = countryName.trim();
```



- To convert the population string to a number, first trim it, then call the `Integer.parseInt` method:

```
int populationValue =
   Integer.parseInt(population.trim());
```

Reading Text Input: Processing Lines

- Occasionally easier to construct a new `Scanner` object to read the characters from a string:

```
Scanner lineScanner = new Scanner(line);
```

- Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

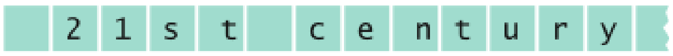
```
String countryName = lineScanner.next();
while (!lineScanner.hasNextInt())
{
   countryName = countryName + " " +
  lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

Reading Text Input: Reading Numbers

- `nextInt` and `nextDouble` methods consume white space and the next number:

```
double value = in.nextDouble();
```

- If there is no number in the input, then a `InputMismatchException` occurs; e.g.



- To avoid exceptions, use the `hasNextDouble` and `hasNextInt` methods to screen the input:

```
if (in.hasNextDouble())
{
   double value = in.nextDouble();
   . . .
}
```

Reading Text Input: Reading Numbers

- `nextInt` and `nextDouble` methods do not consume the white space that follows a number
- Example: file contains student IDs and names in this format:

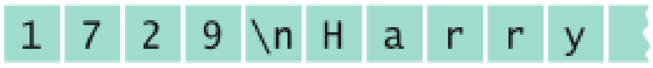
```
1729
Harry Morgan
1730
Diana Lin
. . .
```

- Read the file with these instructions:

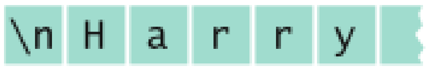
```
while (in.hasNextInt())
{
   int studentID = in.nextInt();
   String name = in.nextLine();
   Process the student ID and name
}
```

Reading Text Input: Reading Numbers

- Initially, the input contains



- After the first call to `nextInt`, the input contains



- The call to `nextLine` reads an empty string! The remedy is to add a call to `nextLine` after reading the ID:

```
int studentID = in.nextInt();
in.nextLine(); // Consume the newline
String name = in.nextLine();
```

Self Check 11.3

Suppose the input contains the characters `6,995.0`. What is the value of `number` and `input` after these statements?

```
int number = in.nextInt();
String input = in.next();
```

Answer: `number` is `6`, `input` is `",995.0"`.

Self Check 11.4

Suppose the input contains the characters `6,995.00 12`. What is the value of `price` and `quantity` after these statements?

```
double price = in.nextDouble();
int quantity = in.nextInt();
```

Answer: `price` is set to `6` because the comma is not considered a part of a floating-point number in Java. Then the call to `nextInt` causes an exception, and `quantity` is not set.

Self Check 11.5

Your input file contains a sequence of numbers, but sometimes a value is not available and marked as N/A. How can you read the numbers and skip over the markers?

Answer: Read them as strings, and convert those strings to numbers that are not equal to N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
   double value = Double.parseDouble(input);
   Process value
}
```

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

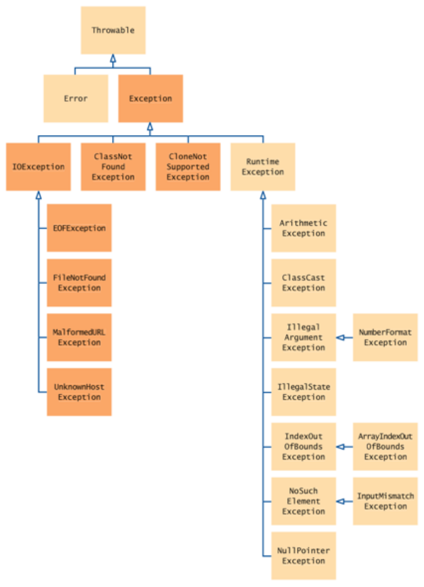


Figure 1 The Hierarchy of Exception Classes

Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

Example

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

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

Self Check 11.6

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 11.7

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

      `NumberFormatException`
      `IllegalArgumentException`
      `NullPointerException`
    - *Example of error:*

      `OutOfMemoryError`

Checked and Unchecked Exceptions

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

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:*

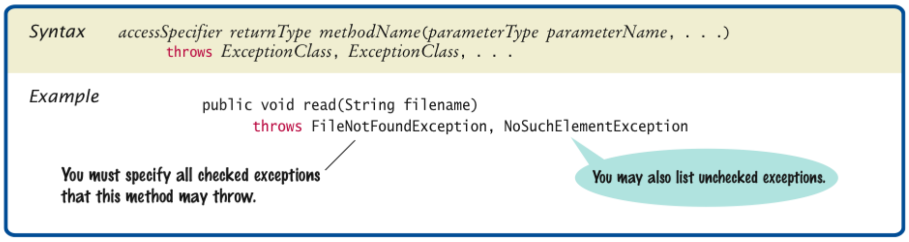
       ```
       public void read(String filename)
          throws IOException, ClassNotFoundException
       ```

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 `throws` Clause



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

Example

```
public void read(String filename)
      throws FileNotFoundException, NoSuchElementException
```

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

You may also list unchecked exceptions.

Self Check 11.8

Suppose a method calls the `Scanner` constructor, which can throw a `FileNotFoundException`, and the `nextInt` method of the `Scanner` class, which can cause a `NoSuchElementException` or `InputMismatchException`. Which exceptions should be included in the `throws` clause?

Answer: You must include the `FileNotFoundException` and you may include the `NoSuchElementException` if you consider it important for documentation purposes. `InputMismatchException` is a subclass of `NoSuchElementException`. It is your choice whether to include it.

Self Check 11.9

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

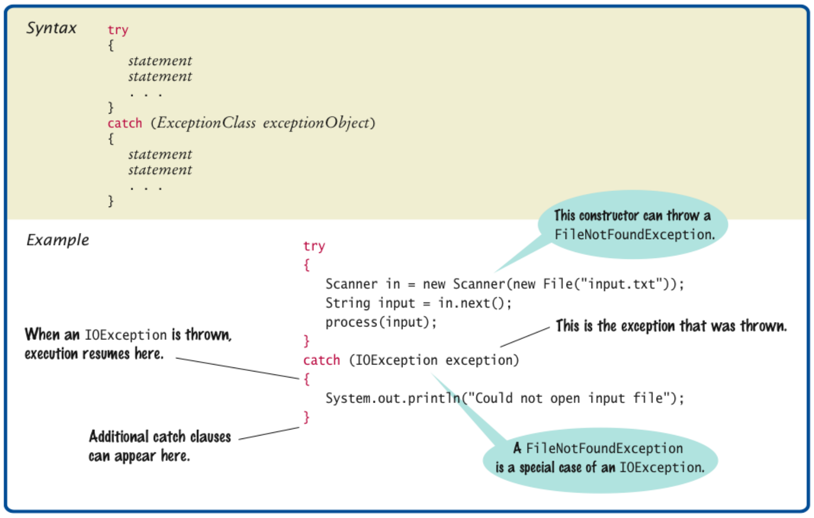
- Example:

```
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 Catching Exceptions



Syntax

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

Example

```
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");
}
```

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.
A FileNotFoundException is a special case of an IOException.

Self Check 11.10

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 11.11

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 above code example. 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"

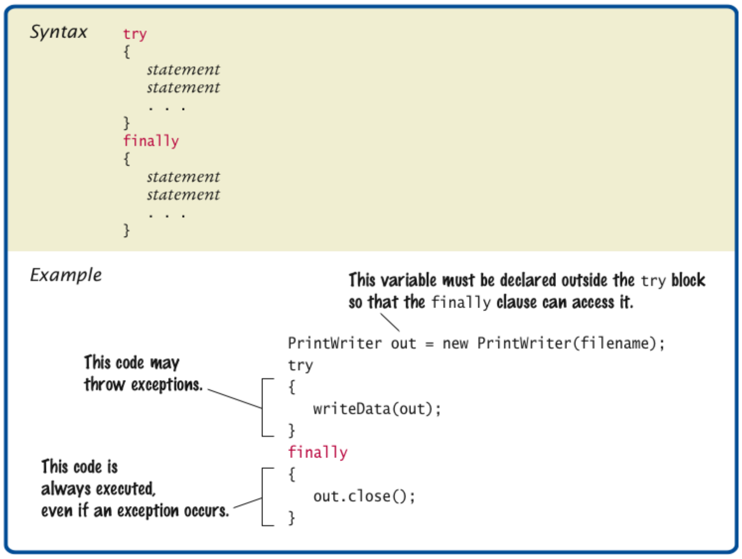
The `finally` Clause

```
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:
  1. *After last statement of* `try` *block*
  2. *After last statement of catch clause, if this* `try` *block caught an exception*
  3. *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 `finally` Clause



Syntax

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

Example

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

This variable must be declared outside the try block so that the finally clause can access it.
This code may throw exceptions.
This code is always executed, even if an exception occurs.

Self Check 11.12

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 `finally` clause could not have closed it.

Self Check 11.13

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

Answer: The `PrintWriter` constructor throws an exception. The assignment to `out` and the `try` block are skipped. The `finally` clause is not executed. This is the correct behavior because `out` has not been initialized.

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 11.14

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 11.15

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.