CHAPTER 7

INPUT/OUTPUT AND EXCEPTION HANDLING

Chapter Goals

- To read and write text files
- To process command line arguments
- To throw and catch exceptions
- To implement programs that propagate checked exceptions

In this chapter, you will learn how to write programs that manipulate text files, a very useful skill for processing real world data.

Contents

- Reading and Writing Text Files
- Text Input and Output
- Command Line Arguments
- Exception Handling
- Application: Handling Input Errors

7.1 Reading and Writing Text Files

- Text Files are very commonly used to store information
  - Both numbers and words can be stored as text
  - They are the most 'portable' types of data files
- The `Scanner` class can be used to read text files
  - We have used it to read from the keyboard
  - Reading from a file requires using the `File` class
- The `PrintWriter` class will be used to write text files
  - Using familiar `print`, `println` and `printf` tools

Text File Input

- Create an object of the `File` class
  - Pass it the name of the file to read in quotes

```
File inputFile = new File("input.txt");
```

- Then create an object of the `Scanner` class
  - Pass the constructor the new `File` object

```
Scanner in = new Scanner(inputFile);
```

- Then use `Scanner` methods such as:
  - `next()`
  - `nextLine()`
  - `hasNextLine()`
  - `hasNext()`
  - `nextDouble()`
  - `nextInt()...`

```
while (in.hasNextLine())
{
  String line = in.nextLine();
  // Process line;
}
```

Text File Output

- Create an object of the `PrintWriter` class
  - Pass it the name of the file to write in quotes

```
PrintWriter out = new PrintWriter("output.txt");
```

- If output.txt exists, it will be emptied
- If output.txt does not exist, it will create an empty file

`PrintWriter` is an enhanced version of `PrintStream`

- `System.out` is a `PrintStream` object!

```
System.out.println("Hello World!");
```

- Then use `PrintWriter` methods such as:
  - `print()`
  - `println()`
  - `printf()`

```
out.println("Hello, World!");
out.printf("Total: %8.2f\n", totalPrice);
```

Closing Files

- You must use the close method before file reading and writing is complete
  - Closing a Scanner

```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process line;
}
in.close();
```

Your text may not be saved to the file until you use the close method!

  - Closing a PrintWriter

```
out.println("Hello, World!");
out.printf("Total: %8.2f\n", totalPrice);
out.close();
```

Exceptions Preview

- One additional issue that we need to tackle:
  - If the input or output file for a Scanner doesn’t exist, a FileNotFoundException occurs when the Scanner object is constructed.
  - The PrintWriter constructor can generate this exception if it cannot open the file for writing.
    - If the name is illegal or the user does not have the authority to create a file in the given location

Exceptions Preview

- Add two words to any method that uses File I/O

```
public static void main(String[] args) throws
   FileNotFoundException
```

  - Until you learn how to handle exceptions yourself

And an important `import` or two..

- Exception classes are part of the `java.io` package
  - Place the `import` directives at the beginning of the source file that will be using File I/O and exceptions

```
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

public class LineNumberer
{
   public void openFile() throws FileNotFoundException
   {
      . . .
   }
}
```

Example: Total.java (1)

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
   This program reads a file with numbers, and writes the numbers to another
   file, lined up in a column and followed by their total.
*/
public class Total
{
   public static void main(String[] args) throws FileNotFoundException
   {
      // Prompt for the input and output file names

      Scanner console = new Scanner(System.in);
      System.out.print("Input file: ");
      String inputFileName = console.next();
      System.out.print("Output file: ");
      String outputFileName = console.next();

      // Construct the Scanner and PrintWriter objects for reading and writing

      File inputFile = new File(inputFileName);
      Scanner in = new Scanner(inputFile);
      PrintWriter out = new PrintWriter(outputFileName);
```

More import statements required! Some examples may use `import java.io.*;`

Note the throws clause

Example: Total.java (2)

```
        // Read the input and write the output

        double total = 0;

        while (in.hasNextDouble())
        {
           double value = in.nextDouble();
           out.printf("%15.2f\n", value);
           total = total + value;
        }

        out.printf("Total: %8.2f\n", total);

        in.close();
        out.close();
    }
}
```

Don't forget to close the files before your program ends.

Common Error 7.1

- Backslashes in File Names
  - When using a String literal for a file name with path information, you need to supply each backslash twice:

```
File inputFile = new File("c:\\homework\\input.dat");
```

  - A single backslash inside a quoted string is the *escape character*, which means the next character is interpreted differently (for example, ‘`\n`’ for a newline character)
  - When a user supplies a filename into a program, the user should not type the backslash twice

Common Error 7.2

- Constructing a `Scanner` with a `String`
  - When you construct a PrintWriter with a String, it writes to a file:

```
PrintWriter out = new PrintWriter("output.txt");
```

  - This does *not* work for a `Scanner` object

```
Scanner in = new Scanner("input.txt"); // Error?
```

  - It does *not* open a file. Instead, it simply reads through the String that you passed (“`input.txt`”)
  - To read from a file, pass `Scanner` a `File` object:

```
Scanner in = new Scanner(new File (“input.txt”));
```

  - or

```
File myFile = new File("input.txt");
Scanner in = new Scanner(myFile);
```

7.2 Text Input and Output

- In the following sections, you will learn how to process text with complex contents, and you will learn how to cope with challenges that often occur with real data.
- Reading Words Example:

Mary had a little lamb

input

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

output

Mary
had
a
little
lamb

Processing Text Input

- There are times when you want to read input by:
  - Each Word
  - Each Line
  - One Number
  - One Character

Processing input is required for almost all types of programs that interact with the user.

- Java provides methods of the `Scanner` and `String` classes to handle each situation
  - It does take some practice to mix them though!

Reading Words

- In the examples so far, we have read text one line at a time
- To read each word one at a time in a loop, use:
  - The `Scanner` object's `hasNext()` method to test if there is another word
  - The `Scanner` object's `next()` method to read one word

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

- Input:

Mary had a little lamb

Output:

```
Mary
had
a
little
lamb
```

White Space

- The `Scanner`’s `next()` method has to decide where a word starts and ends.
- It uses simple rules:
  - It consumes all white space before the first character
  - It then reads characters until the first white space character is found or the end of the input is reached

White Space

- What is whitespace?
  - Characters used to separate:
    - Words
    - Lines

Common White Space

' '	Space
`\n`	NewLine
`\r`	Carriage Return
`\t`	Tab
`\f`	Form Feed

"Mary had a little lamb,\n
her fleece was white as\tsnow"

The useDelimiter Method

- The Scanner class has a method to change the default set of delimiters used to separate words.
  - The useDelimiter method takes a String that lists all of the characters you want to use as delimiters:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

The `useDelimiter` Method

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
```

- You can also pass a String in *regular expression* format inside the String parameter as in the example above.
- `[^A-Za-z]+` says that all characters that `^`not either `A-Z` uppercase letters A through Z or `a-z` lowercase a through z are delimiters.
- Search the Internet to learn more about regular expressions.

Reading Characters

- There are no `hasNextChar()` or `nextChar()` methods of the `Scanner` class
  - Instead, you can set the `Scanner` to use an ‘empty’ delimiter (`""`)

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");

while (in.hasNext())
{
   char ch = in.next().charAt(0);
   // Process each character
}
```

  - `next` returns a one character `String`
  - Use `charAt(0)` to extract the character from the `String` at index 0 to a `char` variable

Classifying Characters

- The Character class provides several useful methods to classify a character:
  - Pass them a char and they return a boolean

```
if ( Character.isDigit(ch) ) …
```

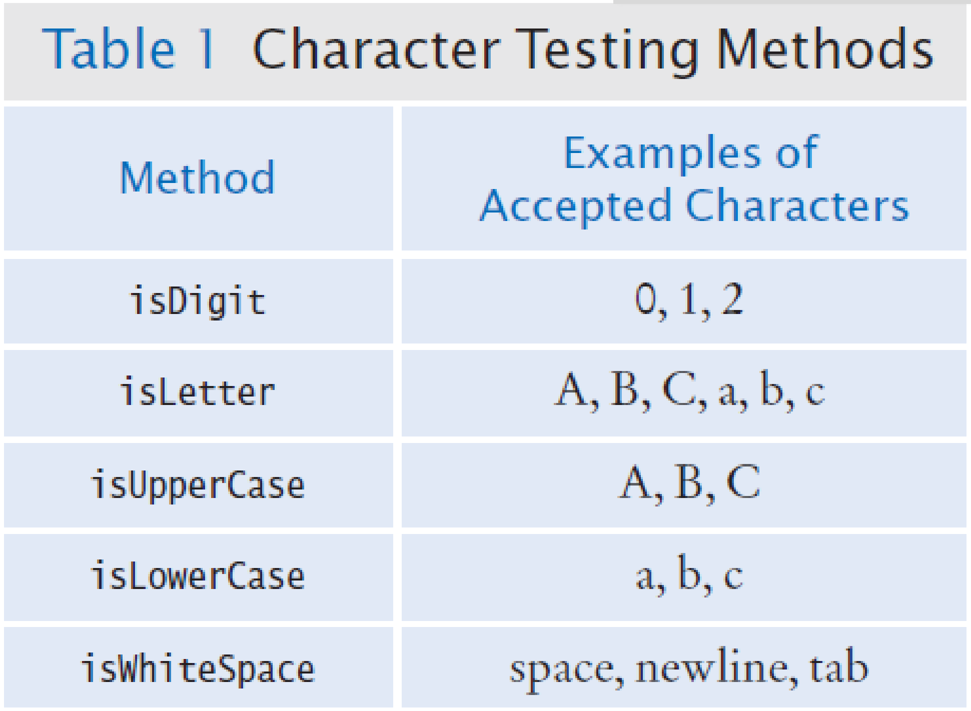


Table 1 Character Testing Methods

Method	Examples of Accepted Characters
isDigit	0, 1, 2
isLetter	A, B, C, a, b, c
isUpperCase	A, B, C
isLowerCase	a, b, c
isWhiteSpace	space, newline, tab

Reading Lines

- Some text files are used as simple databases
  - Each line has a set of related pieces of information
  - This example is complicated by:
    - Some countries use two words
      - "United States"
  - It would be better to read the entire line and process it using powerful `String` class methods

```
China 1330044605
India 1147995898
United States 303824646
```

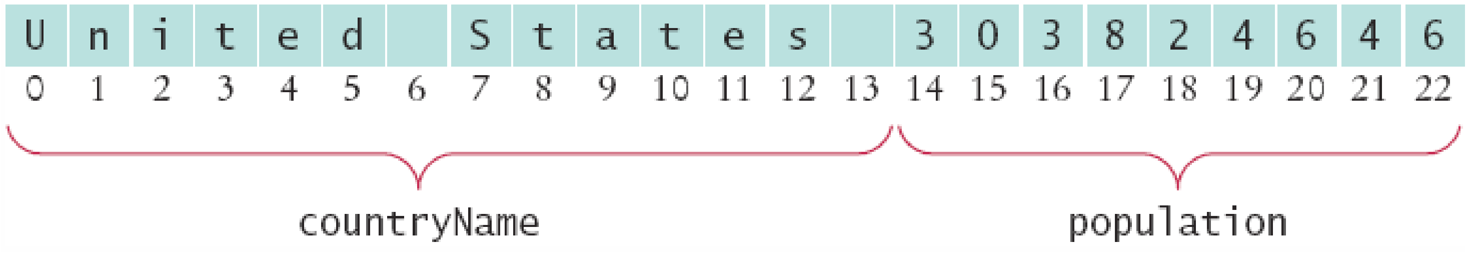
```
while (in.hasNextLine())
{
   String line = in.nextLine();
   // Process each line
}
```

U	n	i	t	e	d		S	t	a	t	e	s		3	0	3	8	2	4	6	4	6
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22

- `nextLine()` reads one line and consumes the ending '`\n`'

Breaking Up Each Line

- Now we need to break up the line into two parts
  - Everything before the first digit is part of the country



  - Get the index of the first digit with `Character.isdigit`

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

Breaking Up Each Line

- Use `String` methods to extract the two parts

```
String countryName = line.substring(0, i);
String population = line.substring(i);
// remove the trailing space in countryName
countryName = countryName.trim();
```

United States

303824646

`trim` removes white space at the beginning and the end.

Or Use Scanner Methods

- Instead of `String` methods, you can sometimes use `Scanner` methods to do the same tasks
  - Read the line into a `String` variable United States 303824646
    - Pass the `String` variable to a new `Scanner` object
  - Use `Scanner` `hasNextInt` to find the numbers
    - If not numbers, use `next` and concatenate words

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

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

Remember the next method consumes white space.

Converting Strings to Numbers

- Strings can contain *digits*, not *numbers*
  - They must be converted to numeric types
  - ‘Wrapper’ classes provide a `parseInt` method

‘3’	‘0’	‘3’	‘8’	‘2’	‘4’	‘6’	‘4’	‘6’

```
String pop = “303824646”;
int populationValue = Integer.parseInt(pop);
```

‘3’	‘.’	‘9’	‘5’

```
String priceString = “3.95”;
int price = Double.parseInt(priceString);
```

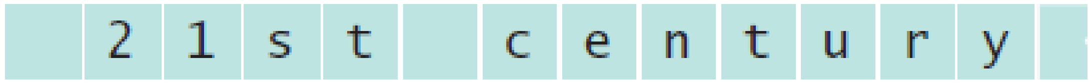
Converting Strings to Numbers

- Caution:
  - The argument must be a string containing only digits without any additional characters. Not even spaces are allowed! So… Use the `trim` method before parsing!

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

Safely Reading Numbers

- Scanner nextInt and nextDouble can get confused



  - If the number is not properly formatted, an "Input Mismatch Exception" occurs
  - Use the hasNextInt and hasNextDouble methods to test your input first

```
if (in.hasNextInt())
{
  int value = in.nextInt();  // safe
}
```

- They will return `true` if digits are present
  - If true, nextInt and nextDouble will return a value
  - If not true, they would 'throw' an 'input mismatch exception'

Reading Other Number Types

- The `Scanner` class has methods to test and read almost all of the primitive types

Data Type	Test Method	Read Method
`byte`	`hasNextByte`	`nextByte`
`short`	`hasNextShort`	`nextShort`
`int`	`hasNextInt`	`nextInt`
`long`	`hasNextLong`	`nextLong`
`float`	`hasNextFloat`	`nextFloat`
`double`	`hasNextDouble`	`nextDouble`
`boolean`	`hasNextBoolean`	`nextBoolean`

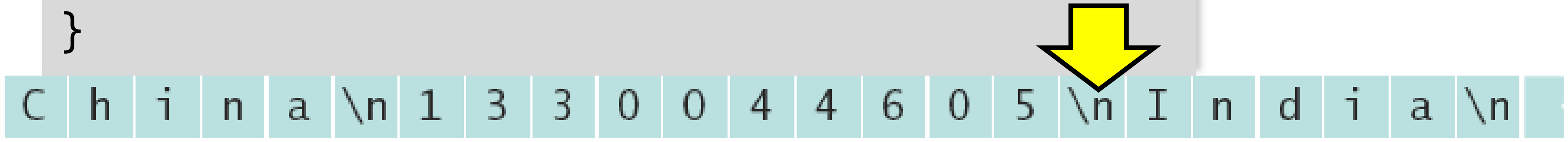
- What is missing?
  - Right, no `char` methods!

Mixing Number, Word and Line Input

- `nextDouble` (and `nextInt`…) do not consume white space following a number
  - This can be an issue when calling `nextLine` after reading a number
  - There is a ‘newline’ at the end of each line
  - After reading 1330044605 with `nextInt`
    - `nextLine` will read until the ‘\n’ (an empty String)

China
1330044605
India

```
while (in.hasNextInt())
{
   String countryName = in.nextLine();
   int population = in.nextInt();
   in.nextLine();    // Consume the newline
}
```



Formatting Output

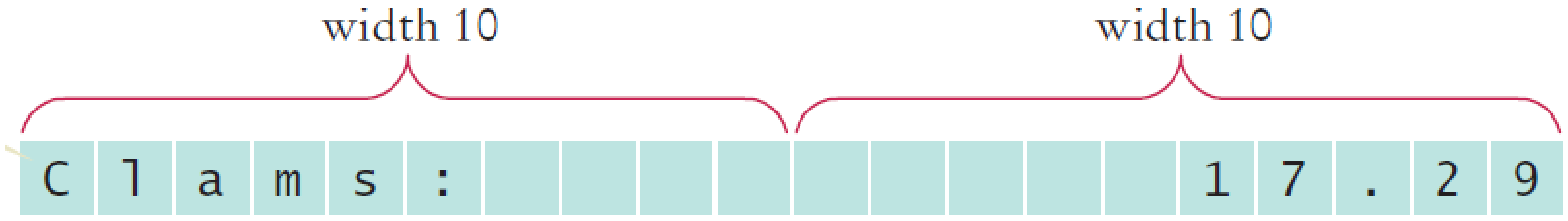
- Advanced `System.out.printf`
  - Can align strings and numbers
  - Can set the field width for each
  - Can left align (default is right)

```
Cookies:       3.20
Linguine:      2.95
Clams:        17.29
```

- Two format specifiers example:

```
System.out.printf("%-10s%10.2f", items[i] + ":", prices[i]);
```

  - `%-10s` : Left justified String, width 10
  - `%10.2f` : Right justified, 2 decimal places, width 10



`printf` Format Specifier

- A format specifier has the following structure:
  - The first character is a %
  - Next, there are optional “flags” that modify the format, such as - to indicate left alignment. See Table 2 for the most common format flags
  - Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers
- The format specifier ends with the format type, such as f for floating-point values or s for strings. See Table 3 for the most important formats

`printf` Format Flags

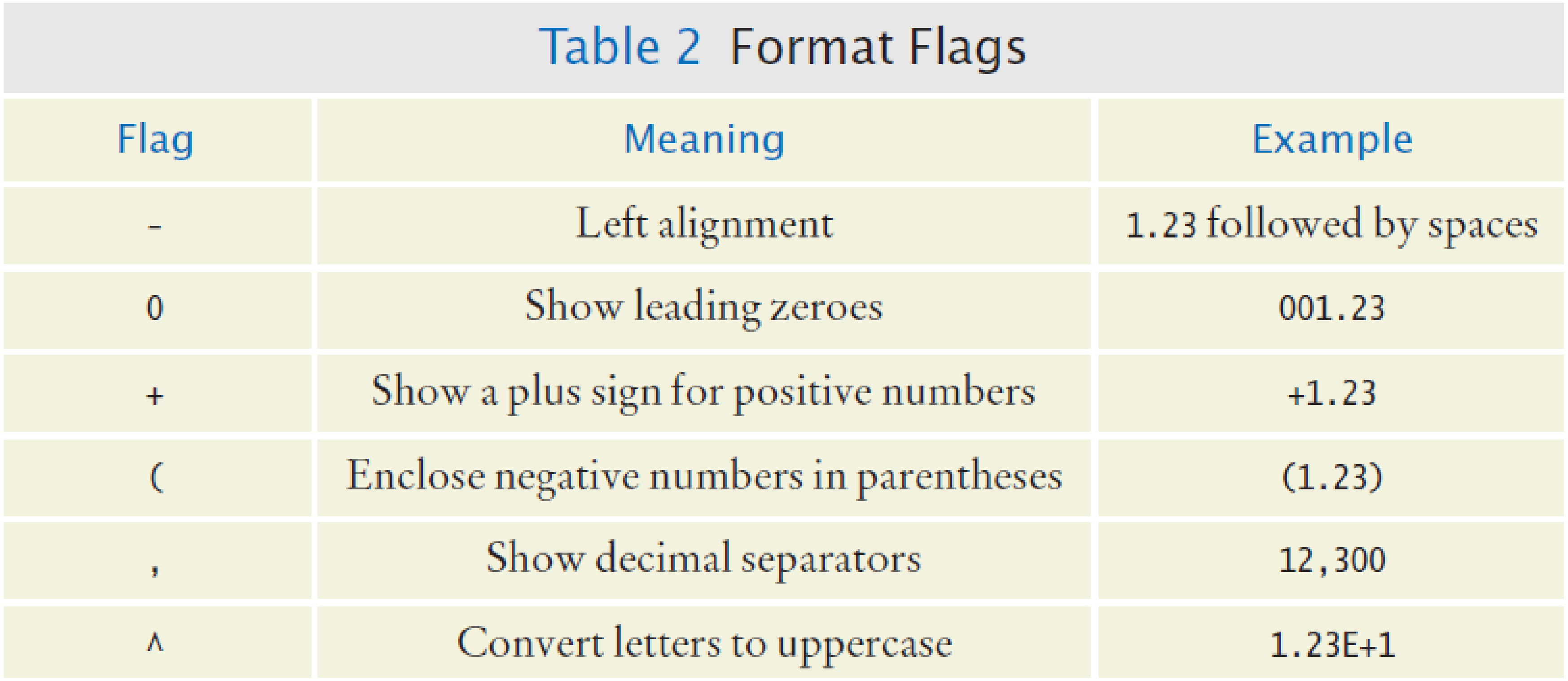


Table 2 Format Flags

Flag	Meaning	Example
`-`	Left alignment	`1.23` followed by spaces
`0`	Show leading zeroes	`001.23`
`+`	Show a plus sign for positive numbers	`+1.23`
`(`	Enclose negative numbers in parentheses	`(1.23)`
`,`	Show decimal separators	`12,300`
`^`	Convert letters to uppercase	`1.23E+1`

printf Format Types

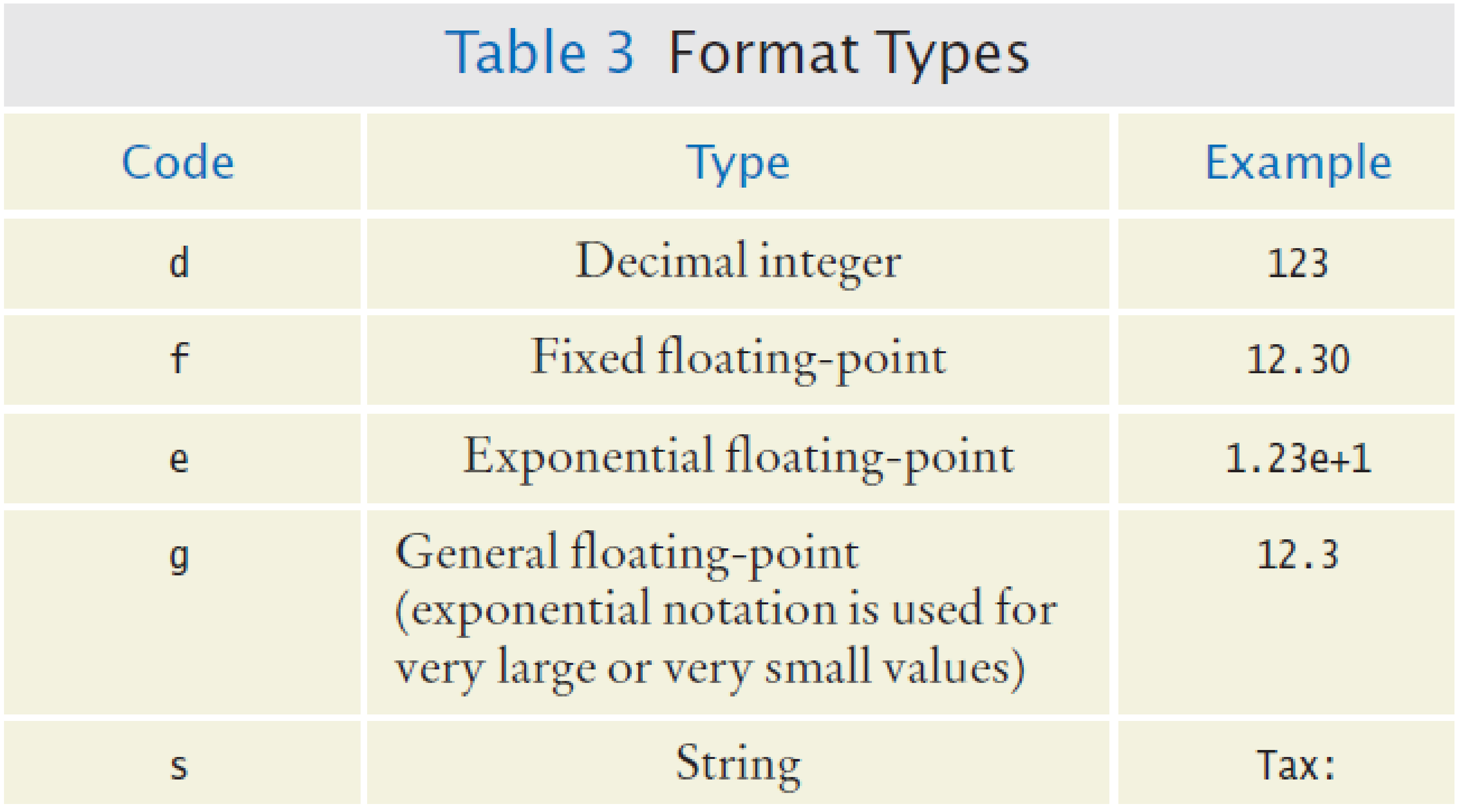


Table 3 Format Types

Code	Type	Example
d	Decimal integer	123
f	Fixed floating-point	12.30
e	Exponential floating-point	1.23e+1
g	General floating-point (exponential notation is used for very large or very small values)	12.3
s	String	Tax:

7.3 Command Line Arguments

- Text based programs can be 'parameterized' by using command line arguments
  - Filename and options are often typed after the program name at a command prompt:

```
>java ProgramClass -v input.dat
```

```
public static void main(String[] args)
```

  - Java provides access to them as an array of `Strings` parameter to the main method named `args`

```
args[0]: "-v"
args[1]: "input.dat"
```

  - The `args.length` variable holds the number of args
  - Options (switches) traditionally begin with a dash '-'

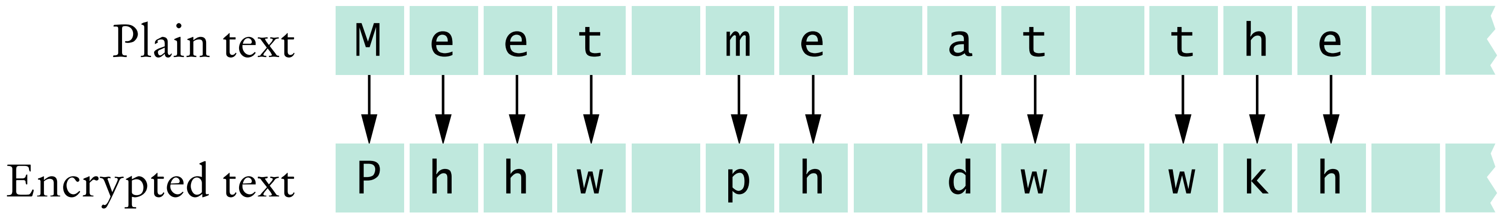
Caesar Cipher Example

- Write a command line program that uses character replacement (Caesar cipher) to:
  1) Encrypt a file provided input and output file names

```
>java CaesarCipher input.txt encrypt.txt
```

  2) Decrypt a file as an option

```
>java CaesarCipher –d encrypt.txt output.txt
```



CaesarCipher.java (1)

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
   This program encrypts a file using the Caesar cipher.
*/
public class CaesarCipher
{
   public static void main(String[] args) throws FileNotFoundException
   {
      final int DEFAULT_KEY = 3;
      int key = DEFAULT_KEY;
      String inFile = "";
      String outFile = "";
      int files = 0; // Number of command line arguments that are files

```

This method uses file I/O and can throw this exception.

CaesarCipher.java (2)

```
      for (int i = 0; i < args.length; i++)
      {
         String arg = args[i];
         if (arg.charAt(0) == '-')
         {
            // It is a command line option

            char option = arg.charAt(1);
            if (option == 'd') { key = -key; }
            else { usage(); return; }
         }
         else
         {
            // It is a file name

            files++;
            if (files == 1) { inFile = arg; }
            else if (files == 2) { outFile = arg; }
         }
      }
      if (files != 2) { usage(); return; }
```

If the switch is present, it is the first argument

Call the usage method to print helpful instructions

CaesarCipher.java (3)

```
      Scanner in = new Scanner(new File(inFile));
      in.useDelimiter(""); // Process individual characters
      PrintWriter out = new PrintWriter(outFile);

      while (in.hasNext())
      {
         char from = in.next().charAt(0);
         char to = encrypt(from, key);
         out.print(to);
      }
      in.close();
      out.close();
   }
```

Process the input file one character at a time

Don’t forget the close the files!

```

   /**
      Prints a message describing proper usage.
   */
   public static void usage()
   {
      System.out.println("Usage: java CaesarCipher [-d] infile outfile");
   }
}
```

Example of a ‘usage’ method

Steps to Processing Text Files

Read two country data files, `worldpop.txt` and `worldarea.txt`.

Write a file `world_pop_density.txt` that contains country names and population densities with the country names aligned left and the numbers aligned right.

```
Afghanistan      50.56
Akrotiri        127.64
Albania         125.91
Algria           14.18
American Samoa  288.92

. . .
```

Steps to Processing Text Files

1) Understand the Processing Task
   -- Process ‘on the go’ or store data and then process?
2) Determine input and output files
3) Choose how you will get file names
4) Choose line, word or character based input processing
   -- If all data is on one line, normally use line input
5) With line-oriented input, extract required data
   -- Examine the line and plan for whitespace, delimiters…
6) Use methods to factor out common tasks

Processing Text Files: Pseudocode

- Step 1: Understand the Task
- While there are more lines to be read

```
    Read a line from each file
    Extract the country name
    population = number following the country name in
       the line from the first file
    area = number following the country name in the line
       from the second file
    If area != 0
        density = population / area
    Print country name and density
```

Afghanistan	50.56
Akrotiri	127.64
Albania	125.91
Algria	14.18
American Samoa	288.92
. . .	

7.4 Exception Handling

- There are two aspects to dealing with run-time program errors:
  1) Detecting Errors

     This is the easy part. You can ‘throw’ an exception

     Use the throw statement to signal an exception

```
if (amount > balance)
{
   // Now what?
}
```

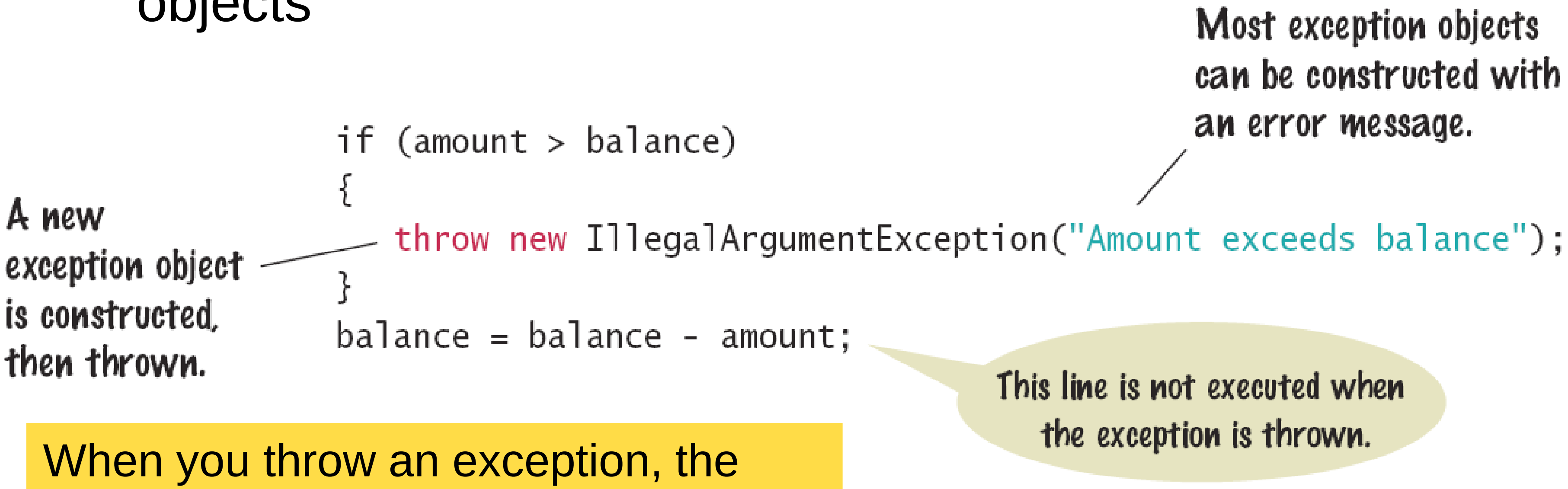
  2) Handling Errors

     This is more complex. You need to ‘catch’ each possible exception and react to it appropriately

- Handling recoverable errors can be done:
  - Simply: exit the program
  - User-friendly: As the user to correct the error

Syntax 7.1: Throwing an Exception

- When you throw an exception, you are throwing an object of an exception class
  - Choose wisely!
  - You can also pass a descriptive String to most exception objects

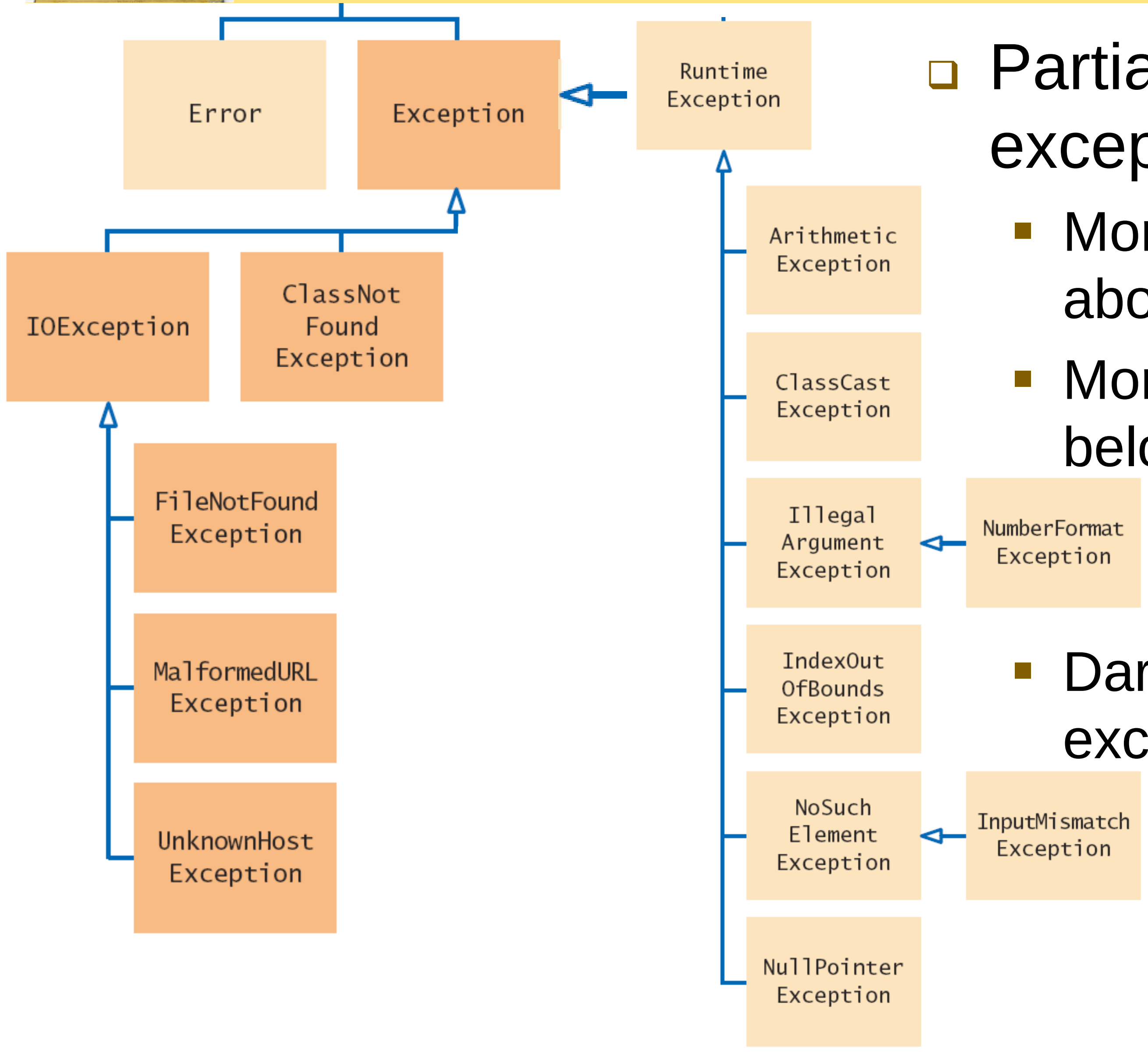


```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

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.

When you throw an exception, the normal control flow is terminated.

Exception Classes



- Partial hierarchy of exception classes
  - More general are above
  - More specific are below
  - Darker are Checked exceptions

Catching Exceptions

- Exceptions that are thrown must be ‘caught’ somewhere in your program

```
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("Input was not a number");
}
```

Surround method calls that can throw exceptions with a ‘try block’.

FileNotFoundException

NoSuchElementException

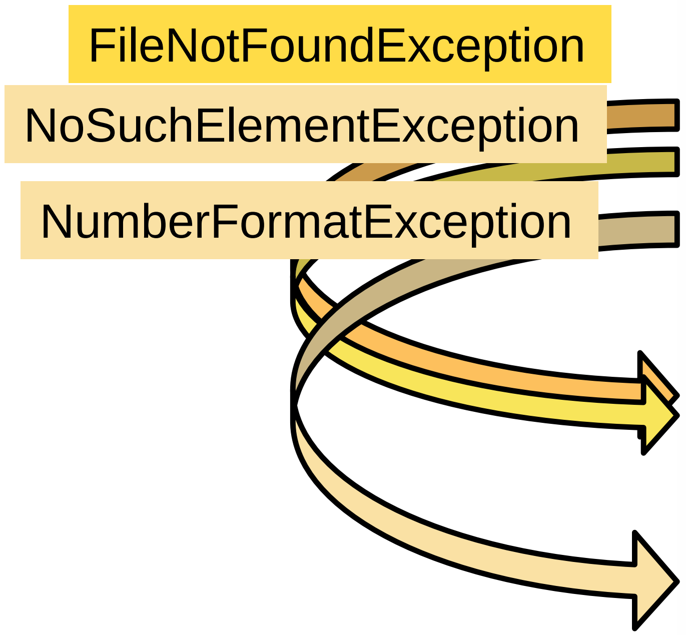
NumberFormatException

Write ‘catch blocks’ for each possible exception.

It is customary to name the exception parameter either ‘`e`’ or ‘`exception`’ in the catch block.

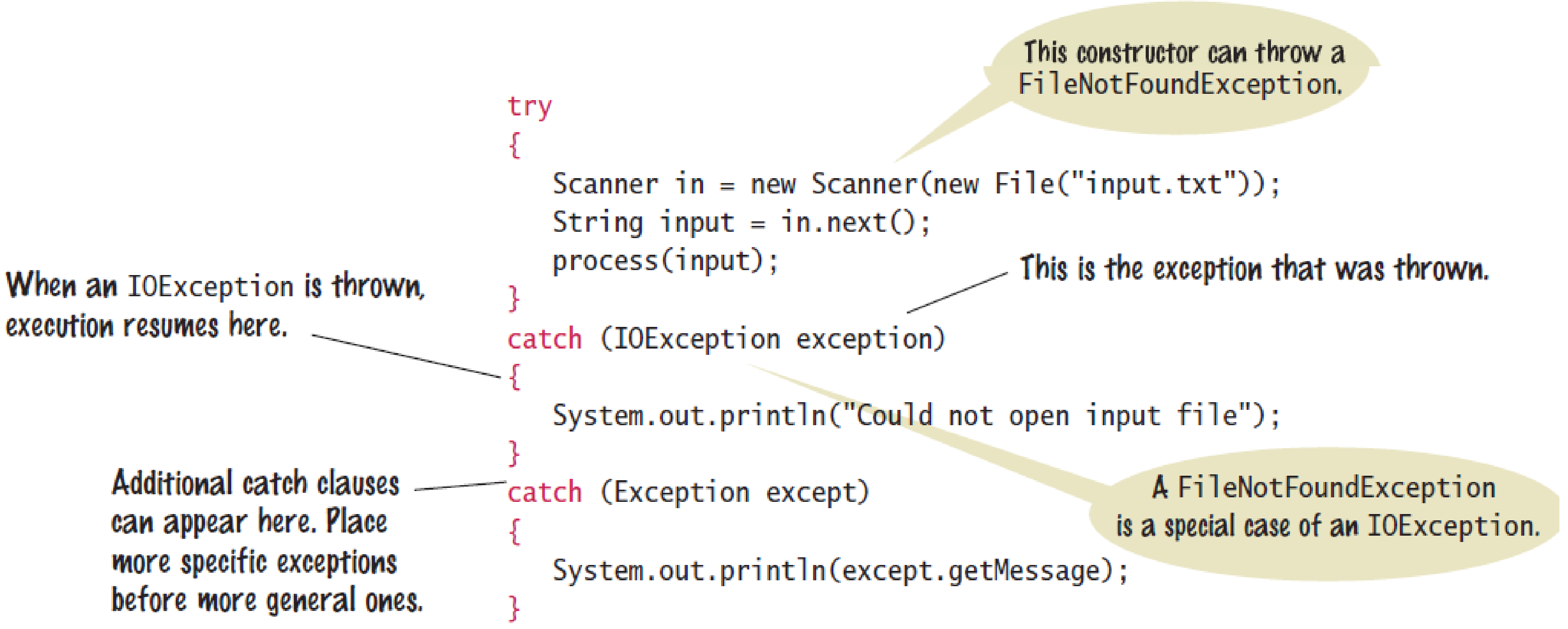
Catching Exceptions

- When an exception is detected, execution ‘jumps’ immediately to the first matching `catch` block
  - `IOException` matches both `FileNotFoundException` and `NoSuchElementException` is not caught



```
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("Input was not a number");
}
```

Syntax 7.2: Catching Exceptions


This constructor can throw a FileNotFoundException.
This is the exception that was thrown.
When an IOException is thrown, execution resumes here.
Additional catch clauses can appear here. Place more specific exceptions before more general ones.
A FileNotFoundException is a special case of an IOException.

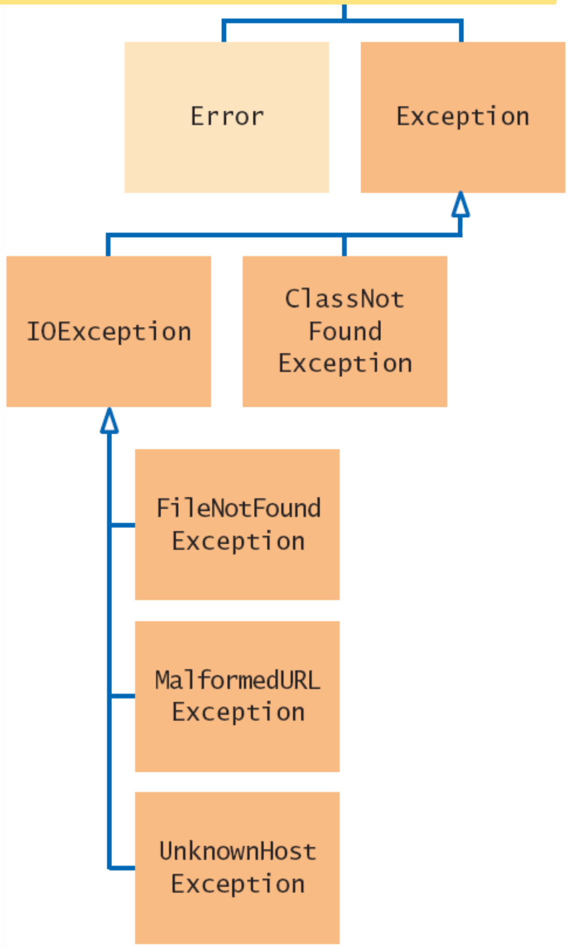
```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
catch (Exception except)
{
   System.out.println(except.getMessage);
}
```

- Some exception handling options:
  - Simply inform the user what is wrong
  - Give the user another chance to correct an input error
  - Print a ‘stack trace’ showing the list of methods called

```
exception.printStackTrace();
```

Checked Exceptions

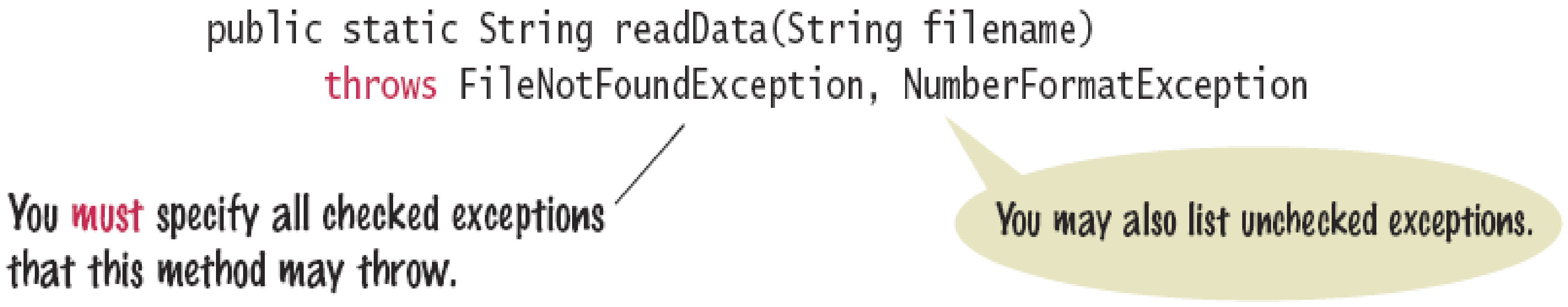
- Throw/catch applies to three types of exceptions:
  - **Error:** Internal Errors
    - not considered here
  - **Unchecked:** RunTime Exceptions
    - Caused by the programmer
    - Compiler **does not check** how you handle them
  - **Checked:** All other exceptions
    - Not the programmer's fault
    - Compiler **checks** to make sure you handle these
    - Shown darker in Exception Classes



Checked exceptions are due to circumstances that the programmer cannot prevent.

Syntax 7.3: The `throws` Clause

- Methods that use other methods that may `throw` exceptions must be declared as such
  - Declare all **checked** exceptions a method throws
  - You may also list **unchecked** exceptions



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

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

You may also list unchecked exceptions.

The `throws` Clause (continued)

- If a method handles a checked exception internally, it will no longer `throw` the exception.
  - The method does not need to declare it in the `throws` clause
- Declaring exceptions in the `throws` clause ‘passes the buck’ to the calling method to handle it or pass it along.

The finally clause

- finally is an optional clause in a try/catch block
  - Used when you need to take some action in a method whether an exception is thrown or not.
    - The finally block is executed in both cases
  - Example: Close a file in a method in all cases

```
public void printOutput(String filename) throws IOException
{
   PrintWriter out = new PrintWriter(filename);
   try
   {
      writeData(out);   // Method may throw an I/O Exception
   }
   finally
   {
      out.close();
   }
}
```

Once a try block is entered, the statements in a finally clause are guaranteed to be executed, whether or not an exception is thrown.

Syntax 7.4: The `finally` Clause

- Code in the `finally` block is always executed once the `try` block has been entered

This variable must be declared outside the `try` block so that the `finally` clause can access it.

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

This code may throw exceptions.

This code is always executed, even if an exception occurs.

Programming Tip 7.1

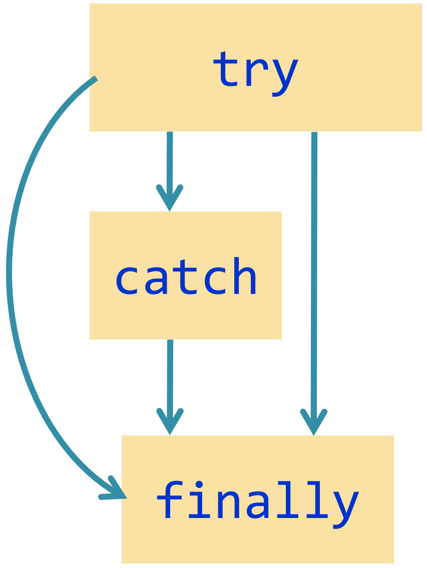
- Throw Early
  - When a method detects a problem that it cannot solve, it is better to throw an exception rather than try to come up with an imperfect fix.
- Catch Late
  - Conversely, a method should only catch an exception if it can really remedy the situation.
  - Otherwise, the best remedy is simply to have the exception propagate to its caller, allowing it to be caught by a competent handler.

Programming Tip 7.2

- Do Not Squelch Exceptions
  - When you call a method that throws a checked exception and you haven’t specified a handler, the compiler complains.
  - It is tempting to write a ‘do-nothing’ catch block to ‘squelch’ the compiler and come back to the code later. **Bad Idea!**
    - Exceptions were designed to transmit problem reports to a competent handler.
    - Installing an incompetent handler simply hides an error condition that could be serious..

Programming Tip 7.3

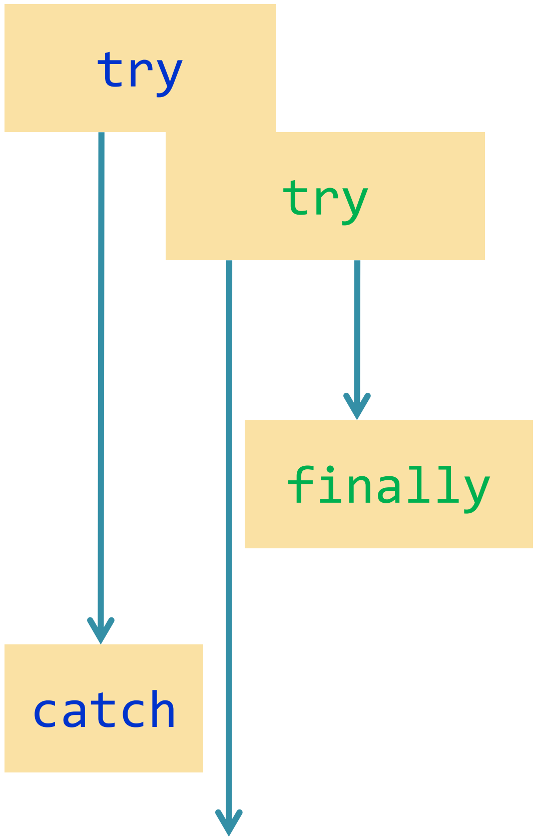
- Do not use `catch` and `finally` in the same `try` block
  - The `finally` clause is executed whenever the try block is exited in any of three ways:
    1. After completing the last statement of the `try` block
    2. After completing the last statement of a `catch` clause, if this try block caught an exception
    3. When an exception was thrown in the `try` block and not caught



Programming Tip 7.3

- It is better to use two (nested) `try` clauses to control the flow

```
try
{
   PrintWriter out = new PrintWriter(filename);
   try
   {        // Write output    }
   finally
   {   out.close(); }   // Close resources
}
catch (IOException exception)
{
   // Handle exception
}
```



7.5 Handling Input Errors

- File Reading Application Example
  - Goal: Read a file of data values
    - First line is the count of values
    - Remaining lines have values
  - Risks:
    - The file may not exist
      - `Scanner` constructor will throw an exception
      - `FileNotFoundException`
    - The file may have data in the wrong format
      - Doesn't start with a count
        - » `NoSuchElementException`
      - Too many items (count is too low)
        - » `IOException`

```
3
1.45
-2.1
0.05
```

Handling Input Errors: main

- Outline for method with all exception handling

```
boolean done = false;
while (!done)
{
   try
   {
      // Prompt user for file name
      double[] data = readFile(filename);    // May throw exceptions
      // Process data
      done = true;
   }
   catch (FileNotFoundException exception)
   {      System.out.println("File not found.");  }
   catch (NoSuchElementException exception)
   {      System.out.println("File contents invalid.");  }
   catch (IOException exception)
   {      exception.printStackTrace();  }
}
```

Handling Input Errors: readFile

- Calls the `Scanner` constructor
- No exception handling (no `catch` clauses)
- `finally` clause closes file in all cases (exception or not)
- `throws IOException` (back to `main`)

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

Handling Input Errors: readData

- No exception handling (no `try` or `catch` clauses)
- `throw` creates an `IOException` object and exits
- unchecked `NoSuchElementException` can occur

```
public static double[] readData(Scanner in) throws IOException
{
   int numberOfValues = in.nextInt();   // NoSuchElementException
   double[] data = new double[numberOfValues];
   for (int i = 0; i < numberOfValues; i++)
   {
      data[i] = in.nextDouble();         // NoSuchElementException
   }
   if (in.hasNext())
   {
      throw new IOException("End of file expected");
   }
   return data;
}
```

Summary: Input/Output

- Use the `Scanner` class for reading text files.
- When writing text files, use the `PrintWriter` class and the `print/println/printf` methods.
- Close all files when you are done processing them.
- Programs that start from the command line receive command line arguments in the main method.

Summary: Processing Text Files

- The `next` method reads a string that is delimited by white space.
- The `Character` class has methods for classifying characters.
- The `nextLine` method reads an entire line.
- If a string contains the digits of a number, you use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.
- Programs that start from the command line receive the command line arguments in the `main` method.

Summary: Exceptions (1)

- To signal an exceptional condition, use the `throw` statement to throw an exception object.
- When you `throw` an exception, processing continues in an exception handler.
- Place statements that can cause an exception inside a `try` block, and the handler inside a `catch` clause.
- Checked exceptions are due to external circumstances that the programmer cannot prevent.
  - The compiler checks that your program handles these exceptions.

Summary: Exceptions (2)

- Add a `throws` clause to a method that can `throw` a checked exception.
- Once a `try` block is entered, the statements in a `finally` clause are guaranteed to be executed, whether or not an exception is thrown.
- `Throw` an exception as soon as a problem is detected.
- `Catch` it only when the problem can be handled.
- When designing a program, ask yourself what kinds of exceptions can occur.
- For each exception, you need to decide which part of your program can competently handle it.