Birkbeck
UNIVERSITY OF LONDON

Introduction to Software Development (ISD)

David Weston and Igor Razgon

Autumn term 2013

Aims of Week 5

- To read in data from text files.
- To write out data to text files.
- Be able to use command line options
- To understand text processing.
- To understand exception handling.
- Be able to implement exception handling and propagate exceptions.

I/O

- I/O stands for Input/Output
- We have already used I/O:
  - Input from keyboard
  - Output to the screen

- File I/O involves reading and writing to files. There are obvious benefits for interacting with files:
  - Input – No need to enter data manually
  - Output – Make permanent record of data

- We shall be looking at only at text files.

Reading Text Input

To read in a text file we need to

1. Import the `File` class and the `FileNotFoundException` class from the package `java.io` by writing this at the start of the program file:

```
import java.io.File;
import java.io.FileNotFoundException;
```

   Alternatively we can import the entire package

```
import java.io.*;
```

Display Contents of a Text File

To open a text file called “input.txt”, construct a File object using the name of the file.

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

To read in the contents of the file we use the Scanner class.

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

while (in.hasNextLine())
{
  String line = in.nextLine();
  System.out.println(line);
}
```

Closing a File

- You must always close a file after you have finished using it.
- use the `close` method from Scanner

- `in.close();`

Why Close a File?

1. To make sure it is closed if a program ends abnormally (it could get damaged if it is left open).

2. A file opened for writing must be closed before it can be opened for reading.
    - Although Java does have a class that opens a file for both reading and writing, it is not used here.

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

Body of Program So Far

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

while (in.hasNextLine())
{
  String line = in.nextLine();
  System.out.println(line);
}
in.close();
```

Exception Handling

- We shall look more closely at exception handling later in this lecture.

- What if the file "input.txt" does not exist?
  - A FileNotFoundException will occur when the Scanner object is constructed

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

  - This will terminate the program should this exception occur.

The Full Program

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public static void main(String[] args) throws FileNotFoundException
{
    File inputFile = new File("input.txt");
    Scanner in = new Scanner(inputFile);
    while (in.hasNextLine())
    {
      String line = in.nextLine();
      System.out.println(line);
    }
   in.close();
}
```

Write to a File

- To write text files we use the PrintWriter class.

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

- Warning: If the file output.txt already exists, it will be immediately overwritten, all data in it will be lost!
- If the file output.txt does not exist then an empty file named "output.txt" will be created.

- To write content to the file, we use methods from the Printwriter class. For example : print, println and printf

Write To file - Example

```
import java.io.PrintWriter;
import java.io.FileNotFoundException;
public static void main(String[] args) throws
FileNotFoundException {
    PrintWriter out = new PrintWriter("output.txt");
    out.println(“My first text file!”);
   out.close();
}
```

- Program logic similar to reading in a file.
- Care needs to be taken regarding the argument for the file `PrintWriter` (a string) and `Scanner` (needs to be a File object).

Why `FileNotFoundException`?

- 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

Input from the Command Line

- Different ways to start a program:

- `BlueJ:` select "`void main(strings[] args)`" menu option
  - By default there is an empty array {}(i.e. no parameters)
  - If you want to pass, say, three parameters, from a command line you would write an array of strings:
  - `{ "one", "2", "three" }`
- On the command line:
  - To run a program called "programClass"
    - `java ProgramClass`
    - `java ProgramClass one 2 three`
- Allows user to run program with extra arguments
- Useful for automating some problems

Command Line Arguments

- To allow for command line arguments write:

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

In your program an array will automatically be created called "args"

- `java ProgramClass -v input.dat`

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

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

Text Processing

- Often the text you wish to write or read has a rich structure.
  - The data might be in the form of a table
  - Distinguish between numeric and text input
  - Comma delimited data items

- Use `Scanner` and `String` methods to deal with these situations

`Scanner` object methods for dealing with one word at a time:

`hasNext()` test if there is another word

`next()` read one word

Reading one word at a time

```
Scanner in = new Scanner(“Here is some text”);
while (in.hasNext())
{
  String input = in.next();
  System.out.println(input);
}
```

- The output is:

```
Here
is
some
text
```

How white spaces are handled

- White space is a term for any character used to separate words and lines.

Common White Space

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

- The `next()` method 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.

Word Separation

- Do not need to be restricted to white space to denote the breaks between words.
- The `useDelimiter` method takes a String that lists all of the characters you want to use as delimiters (i.e. the breaks between words):

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

- "[^A-Za-z]+" is an example of a *Regular Expression .*
  - This example denotes any characters that are neither uppercase nor lowercase
  - In more detail: ^ is not , A-Z uppercase letters A through Z, a-z lowercase a through z.

Reading Individual Characters

In order to process a String one character at a time, use the following procedure:

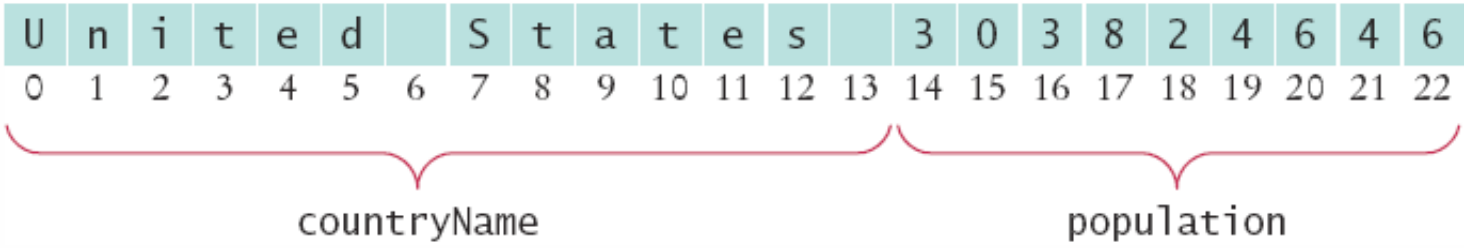
1. Use an empty delimiter.
2. Use charAt(0) to extract the character from the String at index 0 to a char variable

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

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

Reading Individual Characters (cont)

- Traversing one character at a time is useful to find locations within a string. For example consider the following string, where we wish to find the beginning of the number.



- Get the index of the first digit with Character.isDigit

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

- This loop ends with the variable `i` assigned the value 14

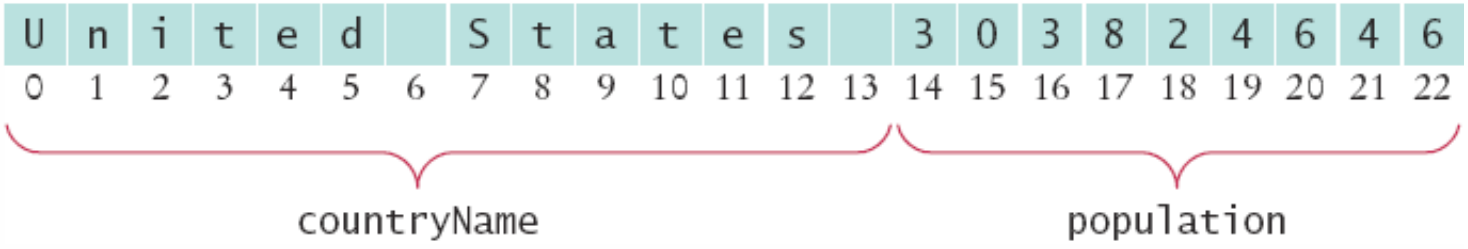
Testing for different types of Character

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

Java for Everyone by Cay Horstmann

Split a String




```
i = 14;
String countryName = line.substring(0, i);
String population = line.substring(i);
```

In order to remove the trailing space in the string *countryName* use `trim()`,this will remove all whitespaces from the beginning and end of the string.

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

Reading lines with Scanner Methods

- You can sometimes use `Scanner` methods to do the same tasks
  1) Read the line into a `String` variable, then pass the `String` variable to a new `Scanner` object
  2) 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.

Reading Other Number 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

Mixing Number, Word and Line Input

- The Reading number methods (`nextDouble`, `nextInt`, etc.) do not consume white space following a number. This can be an issue when calling `nextLine` after reading a number

C	h	i	n	a	\n	1	3	3	0	0	4	4	6	0	5	\n	I	n	d	i	a	\n	

  - There is a 'newline' at the end of each line
  - After reading 1330044605 with `nextInt`
    - `nextLine` will read until the '\n' (an empty String)

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

String to Number Conversion

- A string containing a number is a sequence of characters that happen to be numeric characters. This is different to an actual number. We can convert a string representation of a number into a number using `parseInt`.

```
String pop = “303824646”;
String priceString = “3.95”;

int populationValue = Integer.parseInt(pop);
double price = Double.parseDouble(priceString);
```

The string must only contain digits, not even spaces are allowed. It is often advisable to use the trim method.

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

Formatted Printing

- 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 the slides appended to the end of this lecture 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 the slides appended to the end of this lecture for the most important formats

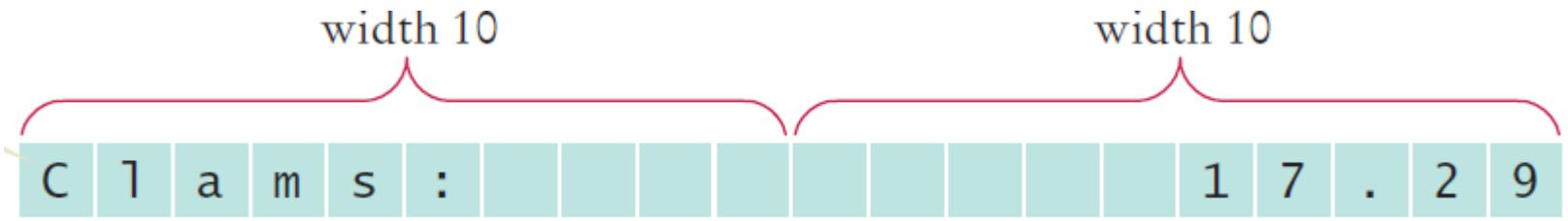
Formatted Printing

- We have already discussed `printf` in previous lectures.

- Can align strings and numbers
- Can set the field width for each
- Can left align (default is right)

```
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



Run-time Program Errors

- Errors can occur in software programs.
- Java has a way of dealing with errors that allows you to separate out your "regular" code from the code that deals with the errors.
- As a consequence, your code will be shorter and easier to understand.

- There are two aspects to dealing with run-time program errors
  1. Detecting errors. This is the easy part
  2. Handling errors. This is more complex, since you will need to consider what is the most appropriate reaction.

Exception Handling - Throwing

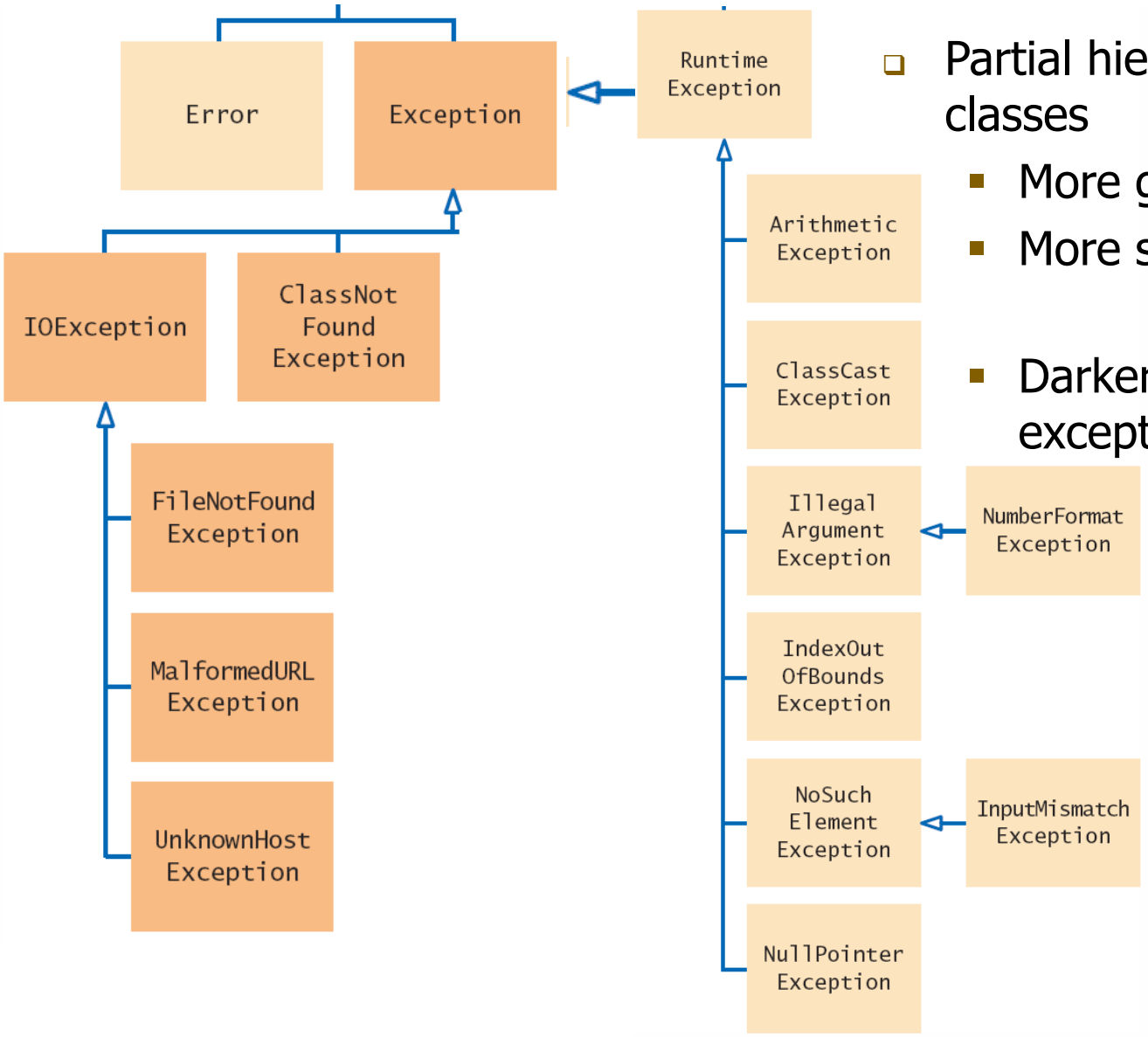
- To detect an error, you throw an *exception* (exceptional event).
- Once an exception is thrown the normal control flow of the program is terminated.

- Create an `IllegalArgumentException` exception object, with an error message and `throw` it.

```
if (amount > balance)
{
        throw new IllegalArgumentException(“Amount
   exceed balance”);
}
balance = balance - amount
```

- Last line not executed if the exception is thrown.

Exception Classes



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

Catching Exceptions

- Once an exception has been thrown, it must be "caught" somewhere else in your program.
  - Place code that can throw an exception into a `try` block
  - Write a `catch` block for each possible exception
- When an exception is detected in the `try` block, the execution jumps immediately to the first matching `catch` block

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   System.out.println(input);
}
// exception parameter named either 'e' or'exception'
catch {IOException e)
{
   System.out.println("Could not open input file");
}
```

Multiple Catch Blocks

```
try{
   Scanner in = new Scanner(new File(“input.txt”));
   String input = in.next();
   System.out.println(input);
}
catch {IOException e){
   System.out.println(“Could not open input file”);
}
catch {NumberFormatException e){
   System.out.println(“Input was not a number”);
}
Catch {Exception e){
   System.out.println(e.getMessage);
}
```

Possible 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 (useful for debugging)

```
e.printStackTrace();
```

- Print error message

```
System.out.println(e.getMessage);
```

Checked versus Unchecked Exceptions

- **Unchecked:** Run-time Exceptions
  - Caused by the programmer, e.g. divide by zero
  - Compiler **does not check** how you handle them

- **Checked:** All other exceptions. These are due to circumstances beyond the control of the programmer.
  - Compiler **checks** to make sure you handle these.
  - Shaded darker in Exception Classes on Slide 34.

Method Declaration using `throw`

- If you create a method that calls other methods that may throw a checked exception, you must declare them as follows:

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

- You may also include unchecked exceptions

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

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

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();
   }
}
```

`Finally` clause scope

- In order for the finally clause to access a variable, it must be declared outside the try block

```
public void printOutput(String filename) throws IOException
{
   PrintWriter out = new PrintWriter(filename); //out declared here
   try
   {
      writeData(out);
   }
   finally
   {
      out.close();                          //so that it can be used here
   }
}
```

- Notice, the method does not catch the exception. It is up to the method calling `printOutput` to deal with it. (This method may in turn propagate the exception to its calling method.)

Using the `finally` clause

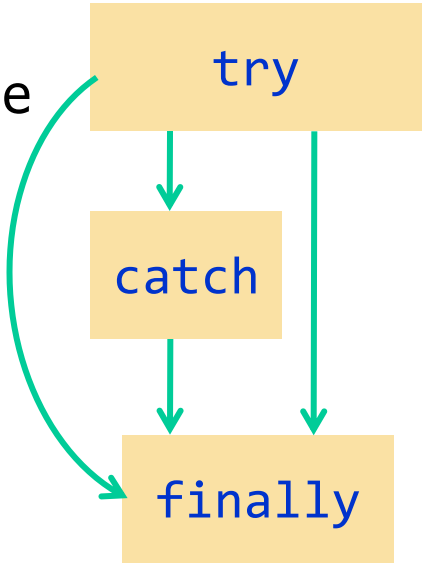
- It is better to use two (nested) `try` clauses to control the flow

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

  }
}
catch (IOException exception)
{
  // Handle exception
}
```

Using the `finally` clause

- 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



Exception Handling

- 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.

Aims of Week 5

- To read in data from text files.
- To write out data to text files.
- Be able to use command line options
- To understand text processing.
- To understand exception handling.
- Be able to implement exception handling and propagate exceptions.

Week 5 Homework

- Complete Lab Sheet 5 – not assessed. Solutions will be posted next week on the BLE.

- Read Chapter 7 of *Java for Everyone* and do the self-check questions as you go.
- Be sure to read Section 7.5 *Application: Handling Input Errors.* It contains a full program that demonstrates many of the error handling concepts.
- If you have time, do some of the review and programming exercises from Chapters 7 of *Java for Everyone*

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

Java for Everyone by Cay Horstmann

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:

Java for Everyone by Cay Horstmann