Chapter 11 – Input/Output and Exception Handling

Dr Kafi Rahman
Assistant Professor @CS
Truman State University

Text Input and Output

- To read just words and discard anything that isn't a letter:
- We can use the Delimiter method of the Scanner class
  - `Scanner in = new Scanner(. . .);`
    `in.useDelimiter("[^A-Za-z]+");`
  - `. . .`
- We shall discuss more about regular expressions later

Text Input and Output - Reading Characters

- To read one character at a time, set the delimiter pattern to the empty string:
  - Scanner in = new Scanner(. . .);
  - in.useDelimiter("");
- Now each call to next returns a string consisting of a single character.
- To process the characters:

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

Text Input and Output - Classifying Characters

The Character class has methods for classifying characters.

Text Input and Output - Reading Lines

- The nextLine method reads a line of input and consumes the newline character at the end of the line:
  - String line = in.nextLine();
- The hasNextLine method returns true if there are more input lines, false when all lines have been read.
- Example: process a file with population data from the CIA Fact Book with lines like this:
  - China 1330044605
  - India 1147995898
  - United States 303824646, etc
- Read each input line into a string

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

Text Input and Output - Reading Lines

- The Character class has methods for classifying characters.

Table 1 Character Testing Methods

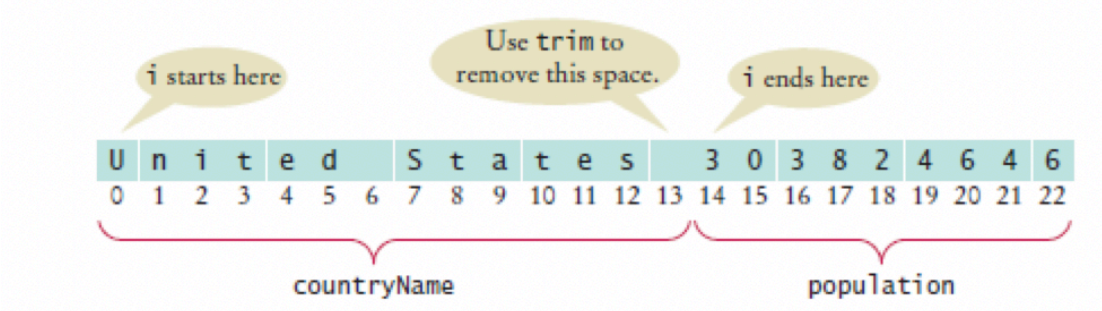
Method	Examples of Accepted Characters
`isDigit`	0, 1, 2
`isLetter`	A, B, C, a, b, c
`isUpperCase`	A, B, C
`isLowerCase`	a, b, c
`isWhiteSpace`	space, newline, tab

Text Input and Output - Reading Lines

- By using the wrapper Character class, we can use the isDigit and isWhitespace methods to find out where the name ends and the number starts.
- To locate the first digit:

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

- To extract the country name and population:
  - String countryName = line.substring(0, i);
  - String population = line.substring(i);
- Use trim to remove spaces at the beginning and end of the string:
  - countryName = countryName.trim();



Text Input and Output - Scanning a String

- Alternatively, it might be easier to construct a new Scanner object to read the characters from a string:
  - Scanner lineScanner = new Scanner(line);

- Then we 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();
```

Text Input and Output - Converting Strings to Numbers

- If a string contains the digits of a number.
- Use the Integer.parseInt or Double.parseDouble method to obtain the number value.
- If the string contains "303824646"
- Use Integer.parseInt method to get the integer value
  - int populationValue = Integer.parseInt(population);
  - // populationValue is the integer 303824646
- If the string contains "3.95"
- Use Double.parseDouble
  - double price = Double.parseDouble(input);
  - // price is the floating-point number 3.95
- In order to remove white spaces in the string, we can use trim:
  - int populationValue = Integer.parseInt(population.trim());

Avoiding Errors When Reading Numbers

- If the input is not a properly formatted number when calling nextInt or nextDouble method then input mismatch exception occurs
- For example, if the input contains characters: " 21st "

- White space is consumed and the word 21st is read.
  - 21st is not a properly formatted number
  - It will cause an input mismatch exception in the nextInt method.
    - programmers can manually remove non digit characters by using isDigit method

- If there is no input at all when you call nextInt or nextDouble,
  - A "no such element exception" occurs.
- To avoid exceptions, always use the hasNextInt method

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

Mixing Number, Word, and Line Input

- The nextInt, nextDouble, and next methods do not consume the white space that follows the number or word.
- This can be a problem if you alternate between calling nextInt/nextDouble/next and nextLine.
- Example: a file contains country names and populations in this format (shown on the right):
- The file is read with these instructions:

```
while (in.hasNextLine())
{
    String countryName = in.nextLine();
    int population = in.nextInt();
    //Process the country name and population.
}
```

China
1330044605
India
1147995898
United States
303824646

Mixing Number, Word, and Line Input

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

- Initial input points to China
- After nextLine, input points to 1330044605
- After nextInt, input still remains at the same line
  - nextInt did not consume the newline character
- Therefore, the second call to nextLine reads an empty string!
- The remedy is to add a call to nextLine after reading the population value and read a dummy value:
  - `String countryName = in.nextLine();`
  - `int population = in.nextInt();`
  - `in.nextLine(); // Consumes the newline`

Formatting Output

- There are additional options for printf method.
- 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

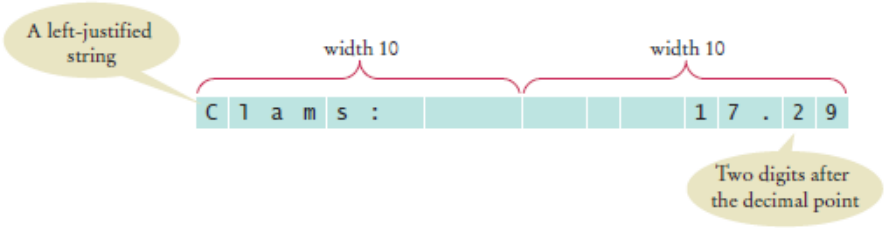
Formatting Output

- For the given program snippet, what would be the output

Format String	Argument	Result
"'%10d'"	1234	' 1234'
"'%-10d'"	1234	'1234 '
"'%010d'"	1234	'0000001234'
"'%,10d'"	1234	' 1,234'
"'%(,10d'"	-1234	' (1,234)'
"%+(,012.2f"	-1200.567f	(001,200.57)

Formatting Output

- To specify left alignment, add a hyphen (-) before the field width:
  - System.out.printf("%-10s%10.2f" , items[i] + ":" , prices[i]);
- There are two format specifiers in the string: "%-10s%10.2f "
- %-10s
  - Formats a left-justified string.
  - Padded with spaces so it becomes ten characters wide
- %10.2f
  - Formats a floating-point number
    - The field that is ten characters wide.
  - Spaces appear to the left and the value to the right
- The output



Formatting Output

- A format specifier has the following structure:
- The first character is a %.
- Next are optional "flags" that modify the format, such as - to indicate left alignment.
- 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.
- 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:

Self Check 11.6

- Suppose the input contains the characters Hello, World!. What are the values of word and input after this code fragment?
  - `String word = in.next();`
  - `String input = in.nextLine();`

Self Check 11.6

- Suppose the input contains the characters Hello, World!. What are the values of word and input after this code fragment?
  - `String word = in.next();`
  - `String input = in.nextLine();`

- Answer: word is "Hello", and input is "World!"

Self Check 11.7

- Suppose the input contains the characters 995.0 Fred. What are the values of number and input after this code fragment?
  - ```
    if (in.hasNextInt()) {
    number = in.nextInt();
    }
    String input = in.next();
    ```
    ```

Self Check 11.7

- Suppose the input contains the characters 995.0 Fred. What are the values of number and input after this code fragment?
  - ```
    if (in.hasNextInt()) {
    number = in.nextInt();
    }
    String input = in.next();
    ```
- Answer: Because 995.0 is not an integer, the call in.hasNextInt() returns false, and the call in.nextInt() is skipped. The value of number stays 0, and input is set to the string "995.0".
    ```

Self Check 11.8

- Suppose the input contains the characters `6E6 6,995.00`. What are the values of `x1` and `x2` after this code fragment?
  - `double x1 = in.nextDouble();`
  - `double x2 = in.nextDouble();`

Self Check 11.8

- Suppose the input contains the characters 6E6 6,995.00. What are the values of x1 and x2 after this code fragment?
  - `double x1 = in.nextDouble();`
  - `double x2 = in.nextDouble();`

- Answer: x1 is set to 6000000. Because a comma is not considered a part of a floating-point number in Java, the second call to nextDouble causes an input mismatch exception and x2 is not set.

Command Line Arguments

- You can run a Java program by typing a command at the prompt in the command shell window
  - Called "invoking the program from the command line"
- With this method, you can add extra information for the program to use called command line arguments
  - Example: start a program with a command line
  - java ProgramClass -v input.dat
- The program receives the strings "-v" and "input.dat" as command line arguments
- Useful for automating tasks
- Your program receives its command line arguments in the args parameter of the main method:
- public static void main(String[] args)
  - In the example, args is an array of length 2, containing the strings
  - args[0]: "-v"
  - args[1]: "input.dat"

Command Line Arguments: an example

```
class CAverage {
  public static void main(String[] args) {
    if(args.length > 0)
     {  double average, sum=0;
        System.out.println("The numbers are ..");
        for(int i=0;i<args.length; i++)
        {  System.out.println(args[i]);
          sum = sum + Integer.parseInt(args[i]);
        }

        System.out.printf("Sum is : %8.2f\n", sum);
        System.out.printf("Average: %8.2f\n", sum / args.length);
        System.out.println("Thanks for using the program\n");
     }
     else {
         System.out.println("No command line parameters provided ..");
     }
  }
}
```

Command Line Arguments: demo

- Let us see it in action

Self Check 11.11

- If the program is invoked with `java ExampleProgram -d file1.txt`, what are the elements of `args`?

Self Check 11.11

- If the program is invoked with java ExampleProgram -d file1.txt, what are the elements of args?

- Answer: args[0] is "-d" and args[1] is "file1.txt"

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
  - For example, 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 can continue if there is 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.

Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

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

A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

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 = "data.txt";
        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 (cont)

- 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 (cont)

- 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 (we should handle it as well)
  - Any exception not caught remains thrown until it is caught by another catch clause.
  - If it is not caught by any catch clause, the program will crash

Thank you

Please let me know if you have any questions.


Questions?