CPS209: COMPUTER SCIENCE II

Input/Output and Exception Handling

- Let's say you call a method:
- x.doSomething(y, z)
- What should happen if the method code detects an error – for example parameter y or paramter z contain an invalid value??

Error Handling

- Traditional approach: Method returns error code
- Problem 1: Calling method forgets to check for error code
  - Failure notification may go undetected
- Problem 2: Calling method may not be able to do anything about failure
  - It must fail too and let its caller worry about it
  - Many method calls would need to be checked

Error Handling

- So, instead of programming for "success":
  - `x.doSomething()`
- You would always be programming for "failure":
  - `if (!x.doSomething()) return false;`
- Or your methods must return error codes:

```
if (x.doSomething() == -99)
    return -99;
else if (x.doSomething() == -88)
    return -88;
```

Throwing Exceptions

- Exceptions:
  - Can't be overlooked
  - Sent directly to an exception handler—not just caller of failed method
- ***Throw*** an *exception object* to signal an exceptional condition
- Example: `IllegalArgumentException`

Example

```
public class BankAccount
{
   double balance;

   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception = new
    IllegalArgumentException("Amount exceeds balance");

         throw exception;
      }
      balance = balance - amount;
   }
   //. . .
}
```

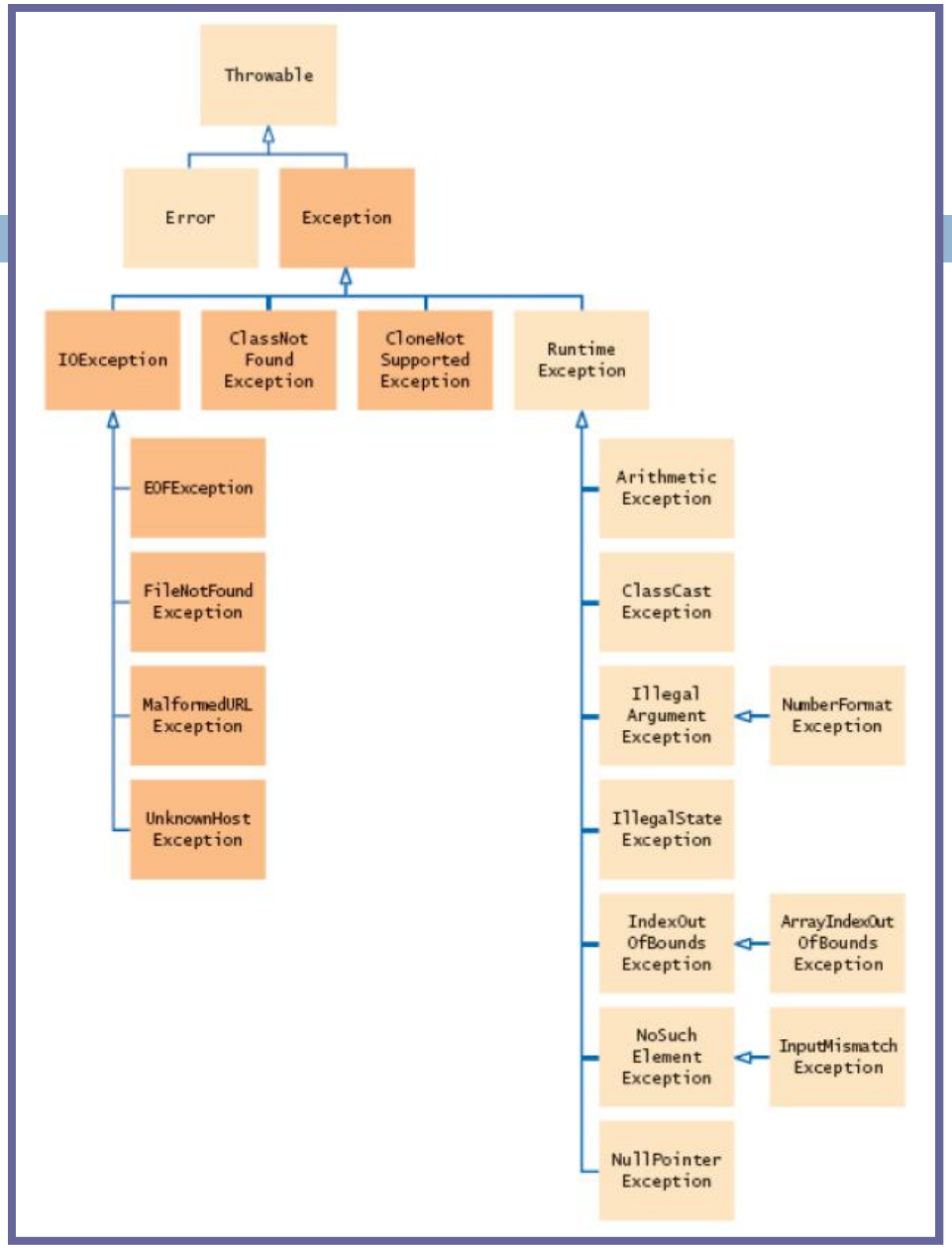
Throwing Exceptions

- No need to store exception object in a temporary variable:

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

- When an exception is thrown, method terminates immediately!!
  - Execution continues in an *exception handler*

Hierarchy of Exceptions Classes



Self-Check

1. How should you modify the `deposit` method to ensure that the balance is never negative?
2. Suppose you construct a new bank account object with a zero balance and then call `withdraw(10)`.

What is the value of `balance` afterwards?

Answers

1. Throw an exception if the amount being deposited is less than zero.
2. 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
    - They are due to external circumstances that the programmer cannot prevent
    - Majority occur when dealing with input and output
    - For example, `IOException`

Checked and Unchecked Exceptions

- Two types of exceptions:
  - Unchecked:
    - Extend the class `RuntimeException or Error`
    - They are typically 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

Input and Output

- Simplest way to read text: use `Scanner` class
- To read from a disk file, construct a `File` object
- Then, use the `File` object to construct a Scanner object:
  - ```
    File inputFile = new File("input.txt");
    Scanner in = new Scanner(inputFile);
    ```
- Use the Scanner methods to read data from file: `next()`, `nextLine()`, `nextInt()`, and `nextDouble()`
    ```

Reading and Writing Text Files: Reading

- A loop to process numbers in the input file:

```
while (in.hasNextDouble())
{
   double value = in.nextDouble();
   // Process value.
   // …
}
```

Reading and Writing Text Files: Writing

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

- You must close a file when you are done processing it:

```
in.close();
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.
  - The `File` constructor can throw a *`FileNotFoundException`*

-
```
void myMethod()
{
  String filename = “myFile.txt”;
  File inputFile = new File(filename);
  Scanner in = new Scanner(inputFile);
  // ...
}
```

Checked Exceptions

- You have two choices:
  1. Handle the exception inside the method or
  2. Tell *compiler* that you want the method to be terminated when the exception occurs (let calling method deal with it)
     - Use `throws` keyword indicating method *can *potentially* throw a checked exception*

- ```
  void myMethod() throws FileNotFoundException
  {
    String filename = “myFile.txt”;
    File inputFile = new File(filename);
    Scanner in = new Scanner(inputFile);
    // ...
  }
  ```
  ```

Checked Exceptions

- For multiple exceptions:

```
void myMethod() throws FileNotFoundException, IOException
{
  String filename = “myFile.txt”;
  File inputFile = new File(filename);
  Scanner in = new Scanner(inputFile);
  // ...
}
```

- Keep in mind inheritance hierarchy:
  If method can throw an IOException and FileNotFoundException, only use IOException
- <u>Better to declare exception than to handle it incompetently</u>

Catching Exceptions

- Install an exception handler with `try/catch` statement
- `try` block contains statements that may cause an exception
- `catch` block contains *handler* code to deal with a particular exception type

Catching Exceptions

```
try
{
   String filename = . . .;
   File inputFile = new File(filename);
   Scanner in = new Scanner(inputFile);
   int i = in.nextInt();

   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (InputMisMatchException exception)
{
   System.out.println("Input was not a number");
}
```

Catching Exceptions

- Statements in <u>try</u> block are executed
- If no exceptions occur, <u>catch</u> clauses are <u>skipped</u>
- If exception of matching type occurs, execution jumps to matching <u>catch</u> clause
- If exception of another type occurs and does not match caught exceptions, it is thrown until it is caught by another <u>try</u> block somewhere else in your program (looks at calling methods)

Catching Exceptions

- `catch (IOException exception){…}`
  - `exception` reference variable 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: General Try Block

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

The `finally` Clause

```
FileReader reader = new FileReader(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   // if exception occurs, execute finally clause
   // before catch clause
   reader.close();
}
```

The `finally` Clause

- Executed when `try` block is exited in any of three ways:
  - After last statement of `try` block
  - After last statement of `catch` clause, if this `try` block caught an exception
  - When an exception was thrown in `try` block and not caught
- Recommendation: don't mix `catch` and `finally` clauses in same `try` block

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

Designing Your Own Exception Types

- Make it an unchecked exception—programmer could have avoided it by calling `getBalance` first
- Extend `RuntimeException` or one of its subclasses
- Supply two constructors
  - Default constructor
  - 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);
   }
}
```

Text Input and Output

- The next method of the Scanner class reads a string that is delimited by white space.
- A loop for processing a file

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

- If the input is "Mary had a little lamb", the loop prints each word on a separate line

```
Mary
Had
A
Little
lamb
```

Text Input and Output

- The next method returns any sequence of characters that is not white space.
- **White space** includes: spaces, tab characters, and the newline characters that separate lines.
- These strings are considered “words” by the next method

  Snow.

  1729

  C++

Text Input and Output

- When next is called:
  - Input characters that are white space are consumed - removed from the input
  - They do not become part of the word
  - The first character that is **not** white space becomes the first character of the word
  - More characters are added until
    1. Either another white space character occurs
    2. Or the end of the input file has been reached
- If the end of the input file is reached before any character was added to the word
  - a “no such element exception” occurs.

Text Input and Output

- To read just words and discard anything that isn't a letter:
  - Call useDelimiter method of the Scanner class

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

- The word separator becomes any character that is **not** a letter.
- Punctuation and numbers are not included in the words returned by the next method.

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.

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

- 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
. . .
```

Text Input and Output – Reading Lines

- Read each input line into a string

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

- Then 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++; }
```

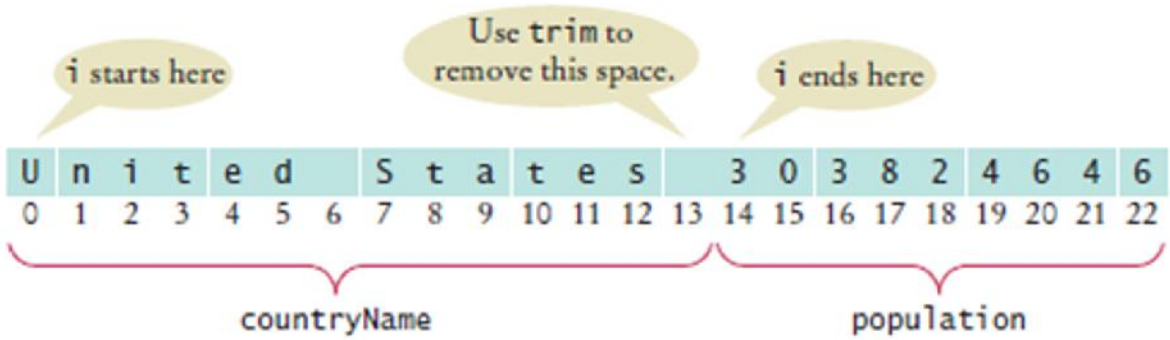
- To extract the country name and population:

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

Text Input and Output – Reading Lines

- Use trim to remove spaces at the beginning and end of string:

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



- Note that the population is stored in a string.

Text Input and Output – Scanning a String

- 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();
```

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
```

- The string must not contain spaces or other non-digits. 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
  - input mismatch exception occurs
- For example, if the input contains characters:

  2 1 s t c e n t u r y

  - White space is consumed and the word 21st is read.
  - 21st is not a properly formatted number
  - Causes an input mismatch exception in the nextInt method.
- If there is no input at all when you call nextInt or nextDouble,
  - A “no such element exception” occurs.
- To avoid exceptions, 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:

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

Mixing Number, Word, and Line Input

- The file is read with these instructions:

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

Mixing Number, Word, and Line Input

- Initial input

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

- Input after first call to nextLine

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

- Input after call to nextInt

  \n I n d i a \n

  - nextInt did **not** consume the newline character

- The second call to nextLine reads an empty string!
- The remedy is to add a call to nextLine after reading the population value:

```
String countryName = in.nextLine();
int population = in.nextInt();
in.nextLine(); // Consume 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
^	Convert letters to uppercase	1.23E+1

Formatting Output

- Example: print a table of items and prices, each stored in an array

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

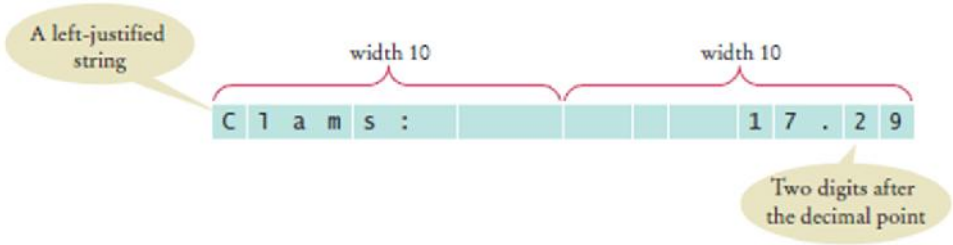
- The item strings line up to the left; the numbers line up to the right.

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: "%-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.

Formatting Output

- 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:

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

Command Line Arguments

- 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"
```