CMPS 200: Introduction to Programming Using JAVA

LECTURE 10 – String Tokenization, File Input / Output, Exceptions

Maurice J. KHABBAZ, Ph.D.

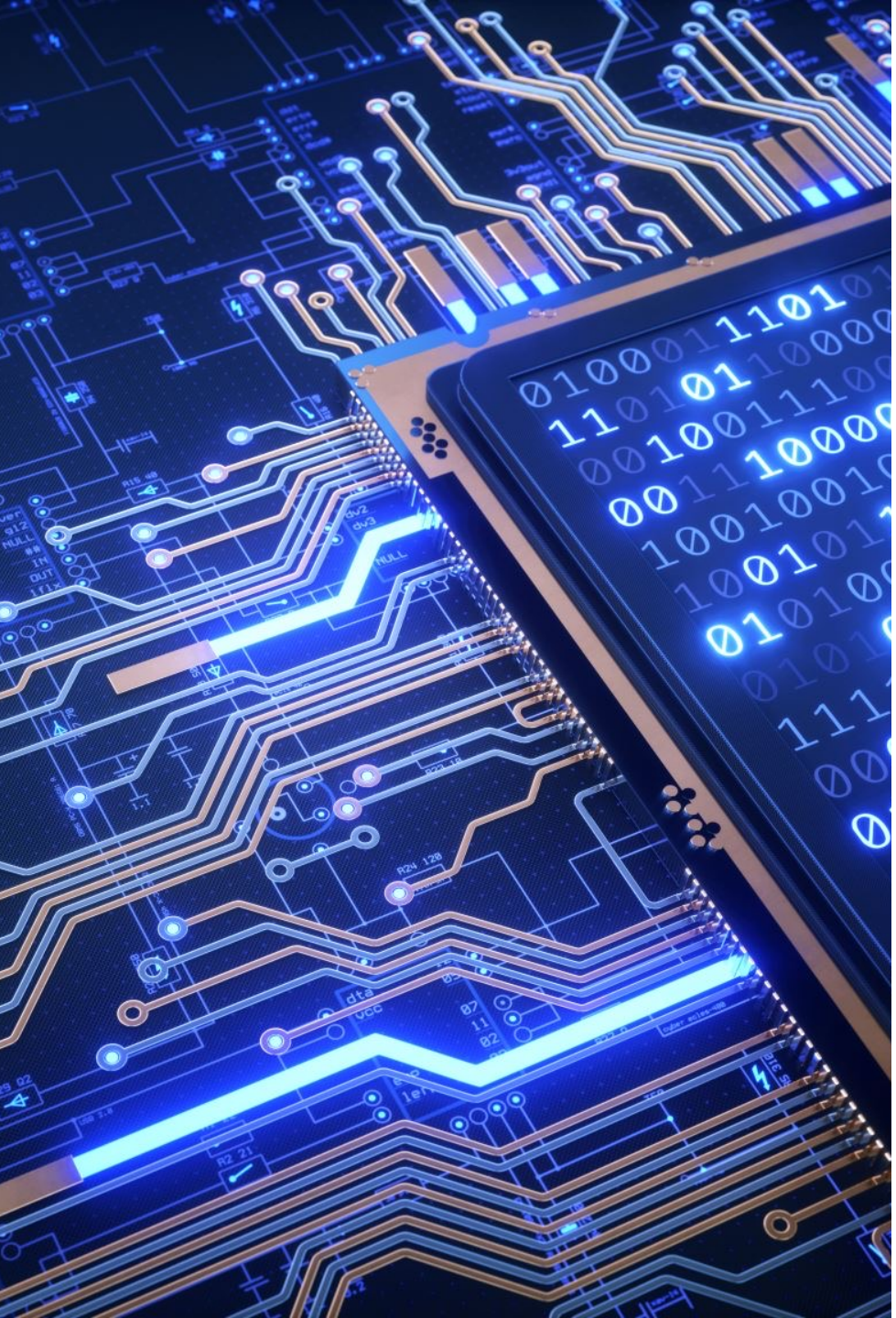
Last Time

Arrays:

- Definition (Reviewed).
- Array Declaration and Usage.
- Bounds Checking and Capacity.
- Arrays Storing Object References.
- Command Line Arguments.
- Arrays as Input/Output Method Parameters.
- Value and Reference Semantics.
- Variable Length Parameter Lists.
- Multidimensional Arrays.
- Introduction to Collections.

Classes:

- `Arrays` class.
- `ArrayList` class.



Today

`Scanner` as an Iterator:

`String` Processing.

`StringTokenizer`

Input / Output.

Exceptions.

Reading From Files.

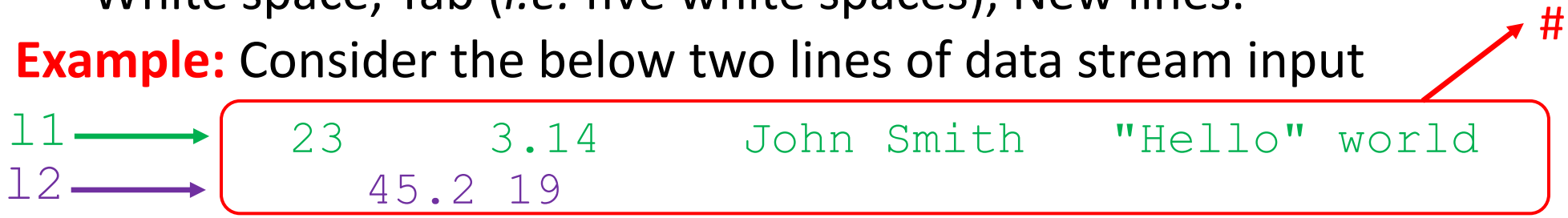
Writing to Files.

File Processing.

Scanner As An Iterator

Consuming Lines of Input

- **Tokens:** elements of the data input stream.
- **Delimiters:** characters that separate tokens from each other.
- A `Scanner` object assumes **default delimiters**:
  - White space, Tab (*i.e.* five white spaces), New lines.
- **Example:** Consider the below two lines of data stream input



`Scanner k` reads the above lines as:

```
String l1 = k.nextLine();
   // 23\t3.14\tJohn Smith\t"Hello" world\n
String l2 = k.nextLine();
   // \t\t45.2 19\n
```

- **REMARK:** Each `\n` character is consumed by not returned.

Token-based `Scanner`

- A `Scanner` object can be set to extract tokens from a `String` object:
  - Similar to taking `String` objects from the standard input stream `System.in`.
- Methods used:

Method	Description
`hasNext()`	returns `true` if there are any more tokens to read from the linked `String` object.
`next()`	returns next token in linked `String` object from cursor to next delimiter.

- **Example:** Print every token in a `String` object `s` on a new line.

```
String s = "Hello! My name is Joe";
Scanner tokenScan = new Scanner(s);
while(tokenScan.hasNext())
    System.out.println(tokenScan.next());
```

OUTPUT

```
Hello!
My
name
is
Joe
```

Tokenize Using: `String` Method `split()` and Arrays

- The `split()` method in the `String` class:
  - Splits a `String` object at particular `String` patterns specified as input parameters.
  - The result will be an array of `String` objects containing the different tokens.
- **Syntax:**

```
String[] <a_name> = <str_name>.split(<pattern>);
```

- **Example:**

```
String str = "Hello! My name is Joe";
String[] spltStr = str.split(" ");
for (String s : spltStr)
    System.out.println(s);
```

OUTPUT

```
Hello!
My
name
is
Joe
```

`StringTokenizer` Class

- Part of the `java.util` package (needs to be imported)
- Allows breaking up a `String` object into tokens.
- Delimiters may be specified either at declaration time or per-token.
- The result is quite similar to the `Scanner` tokenization:
  - Tokens here are not stored as elements of an array for later referencing.
- **Example:**

```
String s = "Sample string to tokenize";
StringTokenizer st = new StringTokenizer(s); // No delimiters.
                                             // Default: white space.
System.out.println("Initial # Tokens: " + st.countTokens());
while(st.hasMoreTokens())
      System.out.println(st.nextToken());
System.out.println("# Tokens left: " + st.countTokens());
```

`StringTokenizer`: A Nice Usecase

- Browsing the **World Wide Web** (`www`):
  - Distributed database of pages linked through the **Hypertext Transport Protocol** (`HTTP`).
  - A webpage consists of objects (*e.g.* `HTML` file, video, photo, applet, … etc)
  - `HTML` file contains several references to various objects:
    - Locations where these objects are stored in the web server's storage space (*e.g.* hard drive, SSD, …).
    - Objects are requested using a **Universal Resource Locator** (`URL`).
    - Typical components of a URL are as follows:

`http`://`www.<website_name>.com`/`<directory>/<object>`

protocol — hostname — pathname

- **Task:** Write a JAVA program called `URLDissector.java` to dissect a `URL`.

SAMPLE OUTPUT

```
Enter a URL: http://www.aub.edu.lb/CMPS/welcome.html
Protocol: http
Hostname: www.aub.edu.lb
Folder: CMPS
Object: welcome.html
```

StringTokenizer: A Nice Usecase

- **Task:** Write a JAVA application called URLDissector.java to dissect the various parts of a URL.
- **Solution:** Use `StringTokenizer` with multiple delimiters:

```
public class URLDissector {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);
        System.out.print("Enter URL: ");
        String url = keyboard.nextLine();
        StringTokenizer st = new StringTokenizer(url, "://");
        ArrayList<String> tokens = new ArrayList<String>();
        while(st.hasMoreTokens()) tokens.add(st.nextToken());
        System.out.println("Protocol: " + tokens.get(0));
        System.out.println("Hostname: " + tokens.get(1));
        System.out.println("Folder  : " + tokens.get(2));
        System.out.println("Object  : " + tokens.get(3));
    }
}
```

Input/Output

Input / Output (I/O)

- The JAVA input/output package is `java.io` (its classes need to be imported).
- File handling is an important part of any application.
- `File` class in `java.io` allows the reading/writing from/to a file.
- **File Declaration Syntax:**

```
File <v_name> = new File("<f_name>.<ext>");
```

- The above declaration **does not create** a new file on the hard drive:
  - To process files on the hard drive, additional methods come to the rescue.

Method name	Description
`<f>.canRead()`	returns whether `<f>` points to a readable file
`<f>.delete()`	removes file pointed to by `<f>` from disk
`<f>.exists()`	returns `true` if the file `<f>` exists on disk; `false` otherwise
`<f>.getName()`	returns the name of the file pointed to by `<f>`
`<f>.length()`	returns number of bytes in the file pointed to by `<f>`
`<f>.renameTo(<f_new>)`	changes `<f>`'s name to that of the file pointed to by `<f_new>`

Example: File Deletion

Problem:

Write a JAVA code snippet that declares a `File` object `f` pointing to an actual file on the hard drive named `example.txt`. If such a file exists and has a size that is larger than `1000` bytes, delete the file.

Solution:

```
File f = new File("example.txt");
if (f.exists() && f.length() > 1000)
     f.delete();
```

Reading files

- One way to read a file is to pass it as a parameter to a `Scanner`.
- **Syntax:**

```
Scanner <name> = new Scanner(new File("<f_name>.<ext>"));
```

- **Example:**

```
File f = new File("mydata.txt");
Scanner inFile = new Scanner(f);

//Alternative shortcut:
Scanner input = new Scanner(new File("mydata.txt"));
```

File Paths

- A **path** is the location of a file on the hard drive.
- **Absolute Path**: specifies a drive or a top (*a.k.a.* root) "/" folder.
  - **Example:** `C:/Documents/smith/hw6/input/data.csv`
  - **Note:** Windows can also use backslashes "\" to separate folders.
- **Relative Path**: does not specify any top/root-level folder:
  - In this case, file identifiers are assumed to be given relatively to the current directory.
  - **Relative Naming Examples:**

```
names.dat              // a data file
input/kinglear.txt. // a txt file in subfolder called input
```

  - **Example:**

```
Scanner input = new Scanner(new File("data/readme.txt"));

// If the JAVA program is stored in the directory C:/CMPS200,
// Scanner will look for C:/CMPS200/data/readme.txt
```

Compile-Time Error Involving Files

- Consider the below JAVA program intended to read a line of text from file:

```
import java.io.File;
import java.util.Scanner;
public class ReadFile {
    public static void main(String[] args) {
        Scanner input = new Scanner(new File("data.txt"));
        String text = input.next();
        System.out.println(text);
    }
}
```

- The above program fails to compile with the following **exception** error:

```
ReadFile.java:6: unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
        Scanner input = new Scanner(new File("data.txt"));
```

Exceptions

- Possible sometimes that instruction execution hit **unexpected conditions**:
  - This is an **exception** … from what was expected and it is **thrown** by a program.
  - It is required to **catch** an exception and specify how to handle/fix it:
    - Otherwise the program will not compile.
- **Examples of Exceptions:**
  - Accessing lists beyond limits:

```
int[] test = {1, 2, 3}; System.out.print(test[4]);
```

ArrayIndexOutOfBoundException

  - Converting an object from one type to another inappropriate type:

```
String s = "hi"; int x = (int) s;
```

Incompatible Types

  - Referencing a non-existing variable:

```
System.out.print(a);
```

Cannot find symbol

  - Mixing data types in operations without coercion:

```
int b = "a"/4;
```

Bad Operand Type

  - Trying to read/write from/to non-existing files:

```
Scanner inFile = new Scanner(f);
```

FileNotFoundException

Working Around Exceptions

- **Q:** What happens when an exception occurs?
- **A:** So far, compiler/interpreter halts program compilation/execution and signals error.
- **Example:**

```
Scanner k = new Scanner(System.in);
System.out.print("Enter x and y: ");
int x = k.nextInt(), y = k.nextInt();
System.out.print("x / y =" + (x / y));
```

- In the above code what can go wrong?
- **Possible Answers:**
  - Cannot convert user input to integers.
  - The user can enter the value `0` for `y` ⇒ `ArithmeticException: / by 0`.

can be avoided using:
- `if … else` statments
- `while` loop until correct input is given

`try ... catch` Statements To Deal With Exceptions

- **`try`** statement defines a code block to be tested for errors during execution.
- **`catch`** statement defines a code block to be executed if exceptions arise.
- `try` and `catch` statements come in matched pairs.
- Can also have nested `try ... catch` statements (same as `if ... else`).
- **Syntax:**

```
try {
    <block of code to try>
} catch (Exception <e>) {
    <block of code to handle exception <e>>
}
```

`try ... catch` Statements To Deal With Exceptions

- **Example:**

```
Scanner k = new Scanner(System.in);
System.out.print("Enter x and y: ");
try {
    int x = k.nextInt(), y = k.nextInt();
    System.out.println("x / y =" + (x / y));
} catch (Exception e) {
    System.out.println("ERROR: non-integers or y = 0.");
}
```

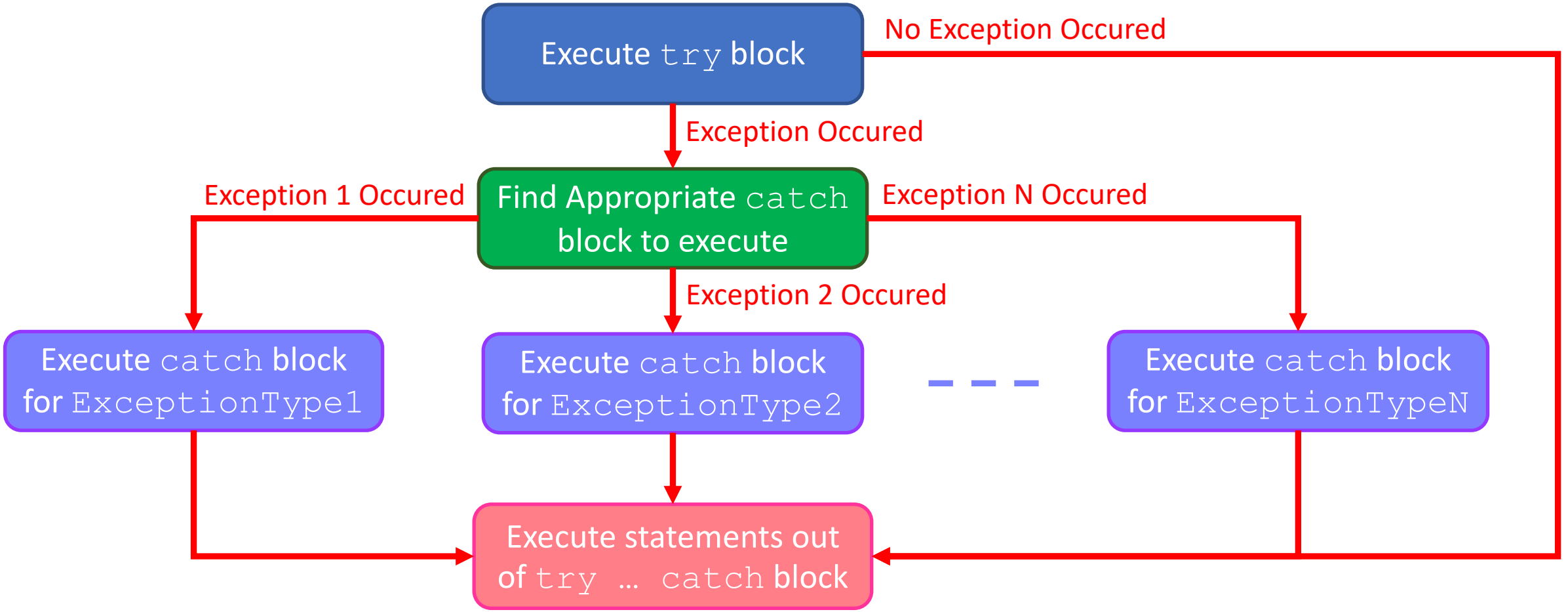
- Exceptions **raised** by any statement in the body of `try`.
- Such exceptions are **handled** by the `catch` statement:
  - Execution continues from within the body of the `catch` statement.

How to differentiate between various types of exceptions?

Differentiating Between Various Exceptions

- A `try` statement can be followed by **multiple `catch` blocks**.
- Each `catch` block will contain a handling procedure for a **precise exception**:
  - This way, exception handling blocks are **customized on a per-exception basis**.
- `catch` blocks must be **ordered** from most specific to most general:
  - Reserved word **`Exception`** designates any exception (most general).
- Only one exception can occur at a time → one of the `catch` blocks will run.

Differentiating Between Various Exceptions



Example 1: Exception Catching Most Specific → Most General

```
public class MultiCatchEx1 {
      public static void main(String[] args) {
            try {
                  int a[] = new int[5]; a[4] = 30 / 0;
            }
            catch (ArithmeticException e) {
                  System.out.print("Division by zero.");
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.print("Index out of bounds.");
            }
            catch (Exception e) {
                  System.out.print("Something went wrong.");
            }
            System.out.print(" Rest of code.");
      }
}
```

Output? **Division by zero. Rest of code.**

Example 2: Exception Catching Most Specific → Most General

```
public class MultiCatchEx2 {
      public static void main(String[] args) {
            try {
                  int a[] = new int[5]; a[5] = 30 / 0;
            }
            catch (ArithmeticException e) {
                  System.out.print("Division by zero.");
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.print("Index out of bounds.");
            }
            catch (Exception e) {
                  System.out.print("Something went wrong.");
            }
            System.out.print(" Rest of code.");
      }
}
```

Output? **Division by zero. Rest of code.**

Example 3: Exception Catching Most Specific → Most General

```
public class MultiCatchEx3 {
      public static void main(String[] args) {
            try {
                  int a[] = new int[5]; a[5] = 30 / 2;
            }
            catch (ArithmeticException e) {
                  System.out.print("Division by zero.");
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.print("Index out of bounds.");
            }
            catch (Exception e) {
                  System.out.print("Something went wrong.");
            }
            System.out.print(" Rest of code.");
      }
}
```

Output? **Index out of bounds. Rest of code.**

Example 4: Exception Catching Most Specific → Most General

```
public class MultiCatchEx4 {
      public static void main(String[] args) {
            try {
                  String s = null; System.out.print(s.length());
            }
            catch (ArithmeticException e) {
                  System.out.print("Division by zero.");
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.print("Index out of bounds.");
            }
            catch (Exception e) {
                  System.out.print("Something went wrong.");
            }
            System.out.print(" Rest of code.");
      }
}
```

Causes a `NullPointerException` (not explicitly caught)

Invokes `catch` block corresponding to the general `Exception`

Output? **Something went wrong. Rest of code.**

Example 5: Not Maintaining Order of Exceptions

```
public class MultiCatchEx5 {
      public static void main(String[] args) {
            try {
                  int[] a = new int[5]; a[5] = 30 / 0;
            }
            catch (Exception e) {
                  System.out.print("Something went wrong.");
            }
            catch (ArithmeticException e) {
                  System.out.print("Division by zero.");
            }
            catch (ArrayIndexOutOfBoundsException e) {
                  System.out.print("Index out of bounds.");
            }
            System.out.print(" Rest of code.");
      }
}
```

Output? **Compile-time Error**

Exercise 1: Sanity Check

Explain how the below code snippet will be executed for different inputs

```
Scanner keyboard = new Scanner(System.in);
try{
      System.out.print("Enter x: "); int x = keyboard.nextInt();
      System.out.print("Enter y: "); int y = keyboard.nextInt();
      System.out.println("So far so good");
      System.out.println( "x / y =" + (x / y));
      System.out.println("All Good!");
} catch (Exception e) {
      System.out.println("Something went wrong!!");
}
System.out.println("Done!");
```

Another Complete Example

```
Scanner k = new Scanner(System.in);
while(true) {
      try {
            System.out.print("Enter an integer n: ");
            n = k.nextInt();
            break;
      } catch (InputMismatchException e) {
            System.out.println("ERROR: n not integer. Try again!");
      }
System.out.println("Correct input of an integer!");
```

Scanner-generated exception (revisited later)

- The above code will keep looping and asking for an input until an integer is entered.

One Additional Clause For Handling Exceptions

- **`finally` Clause:**
  - A clause whose body is **always executed** after `try ... catch` regardless of whether or not exceptions were raised.

- **Example:**

```
try {
        int[] nums = {1, 2, 3};
        System.out.println(myNumbers[10]);
} catch (Exception e) {
        System.out.print("Error Occurred.");
} finally {
        System.out.print("Executing \"finally\" clause.");
}
```

Output? **Error Occurred. Executing "finally" clause.**

The **throws** clause

- **throws:**
  - reserved word injected at the end of a method's header.
  - It is followed by an exception type, <e_type>.
  - All this states that the method may generate an exception but will not handle it.
  - It is like saying:

    "I hereby announce that this method might throw an exception, and I accept the consequences if this happens."

- **Syntax:**

```
public static <type> <m_name>(<params>) throws <e_type> {
```

- **Example:**

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

The **throws** clause

- **Example:** Consider the below JAVA `main()` method

```
public class ReadFile {
    public static void main(String[] args) {
        File f = new File("testing.txt"); // this file may
        ...                               // not exist.
    }
}
```

  - Attempting to compile and execute the above code will result in a compile time error:

```
unreported exception java.io.FileNotFoundException;
must be caught or declared to be thrown
```

  - Adding a throws clause at the end of `main()` method header will lead to:
    - A successful compilation.
    - A possible run-time **`FileNotFoundException`** if the file does not exist:
      - The program will stop running (`throws` declares the exception but does not handle it).

Tokens in a File

- Assume that an input file contains the following data:

```
23 3.14
    "John Smith"     -1.2
```

- A `Scanner` can interpret the tokens as follows:

Token	Possible Types
23	int, double, String
3.14	double, String
"John	String
Smith"	String
-1.2	double

- The `Scanner` will view all input as **a stream of characters**:

input cursor designating `Scanner` current position →

```
\t23 3.14\n\t\t"John Smith"\t\t-1.2\n
^
```

Consumuing Input File Tokens

- Reading tokens and advancing the cursor:
  - calling `Scanner`'s `next()` method(s).

input cursor designating `Scanner` current position

- **Example:**
  - Calling `nextInt()` reads `23` and advances cursor past this value:

  - Now, calling `nextDouble()` reads `3.14` and advances cursor past this value.

Exercice 2: Reading/Processing File Content

Consider a file called `weather.txt` that contains daily temperature information as follows:

```
16.2   23.5
   19.1 7.4  22.8

18.5  -1.8 14.9
```

Write a JAVA application that reads this file's data and prints the change in daily temperature producing the output to the right.

OUTPUT TO THE SCREEN

```
16.2 to 23.5, change =   7.3
23.5 to 19.1, change =  -4.4
19.1 to  7.4, change = -11.7
 7.4 to 22.8, change =  15.4
22.8 to 18.5, change =  -4.3
18.5 to -1.8, change = -20.3
-1.8 to 14.9, change =  16.7
```

Remark:
There are exactly 7 change entries.

Exercise 2: Solution

```
// Displays changes in temperature from data in an input file.
import java.io.File;
import java.util.Scanner;

public class RadingTemperatures {
    public static void main(String[] args) throws FileNotFoundException {
        File f = new File("weather.txt");
        Scanner fs = new Scanner(f);
        double prev = fs.nextDouble();
        for (int i = 1; i <= 7; i++) {
            double next = fs.nextDouble();
            System.out.println(prev + " to " + next + ", change = " + (next - prev));
            prev = next;
        }
    }
}
```

Reading The Content Of An Entire File

- **Limitation:** Exercise 2 Solution is specific and works only for seven entries.
- **Requirement:** program regardless of the number of entries in the file.
- **Observations:**
  - **Q:** What happens if the file has more entries than needed (or at least expected)?
  - **A:** No entry beyond the seventh will be read.
  - **Q:** What happens if the file has fewer entries than required?
  - **A:** The program will crash!
- **Example:**
  - Output from a file with just 3 temperature values `16.2`, `23.5` and `19.1`:

```
16.2 to 23.5, change =  7.3
23.5 to 19.1, change = -4.4
Exception in thread "main” java.util.NoSuchElementException
    at java.util.Scanner.throwFor(Scanner.java:838)
    at java.util.Scanner.next(Scanner.java:1347)
    at Temperatures.main(Temperatures.java:12)
```

`Scanner` Exceptions

- **`NoSuchElementException`:** Occurs when reading past the end of an input.
- **`InputMismatchException`:** Occurs when reading wrong token type:
  - **Example:** reading "`hi`" as an `int`.
- **Requirement:** find and fix these exceptions.
- `Scanner` methods useful for this purpose:

Method	Description
`hasNext()`	returns `true` if there is a next token
`hasNextInt()`	returns `true` if next token exists and can be read as `int`
`hasNextDouble()`	returns `true` if next token exists and can be read as a `double`

- Above methods **do not consume input**:
  - They just **give information** about the existence of next token and its type.
  - Help in avoiding crashes (*i.e.* run-time errors halting execution).

Examples: Using `hasNext()` Methods

- **To avoid type mismatches:**

```
Scanner k = new Scanner(System.in);
System.out.print("How old are you? ");
if (k.hasNextInt()) {
    int age = k.nextInt();          // will not crash!
    System.out.println("Wow, " + age + " is old!");
} else
    System.out.println("ERROR: Expected an int.");
```

- **To avoid reading past the end of a file:**

```
Scanner fs = new Scanner(new File("essay.txt"));
while (fs.hasNext()) {
    String line = fs.next();    // will not crash!
    System.out.println(line);
}
```

Exercice 3: Processing Entire File Content

Modify the `ReadingTemperatures.java` program of Exercise 2 to process the entire `weather.txt` file regardless of the number of entries it has and produce an output such as that shown to the right.

OUTPUT TO THE SCREEN

```
16.2 to 23.5, change =   7.3
23.5 to 19.1, change =  -4.4
19.1 to  7.4, change = -11.7
 7.4 to 22.8, change =  15.4
22.8 to 18.5, change =  -4.3
18.5 to -1.8, change = -20.3
-1.8 to 14.9, change =  16.7

...
```

```
import java.io.*; import java.util.*;
public class ReadingTemperatures {
  public static void main(String[] args)
                throws FileNotFoundException {
    File f = new File("weather.txt");
    Scanner fs = new Scanner(f);
    double prev = fs.nextDouble();
    while (fs.hasNextDouble()) {
        double next = fs.nextDouble();
        System.out.println(prev + " to " + next + ", change = " + (next - prev));
        prev = next;
    }
  }
}
```

Exercice 4: Processing Entire File Content

Modify the `ReadingTemperatures.java` program of Exercise 3 to process the entire `weather.txt` file regardless of the number of entries as well as the existence of non-numeric tokens (these need to be skipped).

SAMPLE INPUT FILE

```
16.2   23.5
Tuesday   19.1   Wed 7.4
THURS. TEMP: 22.8

18.5  -1.8  <-- Marty here is
my data!  --Kim
   14.9 :-)
```

```
import java.io.*; import java.util.*;
public class ReadingTemperatures {
  public static void main(String[] args)
                throws FileNotFoundException {
    File f = new File("weather.txt");
    Scanner fs = new Scanner(f);
    double prev = fs.nextDouble();
    while(fs.hasNextDouble()) {              // Processes pronenumeric tokens.
      if (input.hasNextDouble()) {
        double next = fs.nextDouble();
        double next = input.nextDouble();
        System.out.println(prev + " to " + next + ", change = " + (next - prev));
          System.out.println(prev + " to " + next + ", change = " + (next - prev));
        prev = next;
          prev = next;
    }  } else fs.next();                     // throw away unwanted token.
  }  }}} // body closure done this way due to space limitation.
}
```

Exercise 5: Electoral Polls

Write a JAVA program called ElectroralPolls.java that reads a file polls.txt of electoral poll data.
The file format (yellow) followed by some samples (green) are as follows:

```
State   Trump%   Biden%   ElectoralVotes   Pollster
CT      56       31       7                Oct U. of Connecticut
NE      37       56       5                Sep Rasmussen
AZ      41       49       10               Oct Northern Arizona U.
```

The program should print how many electoral votes each candidate leads in, and who is leading overall in the polls.

Sample Output:

```
Trump : 214 votes
Biden : 257 votes
```

Exercise 5: Solution

```
// Computes leader in presidential polls, based on input file such as:
// AK 42 53 3 Oct Ivan Moore Research

import java.io.File;
import java.util.Scanner;
public class Elections {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner fs = new Scanner(new File("polls.txt"));
        int trumpVotes = 0, bidenVotes = 0;
        while (fs.hasNext()) {
            if (fs.hasNextInt()) {
                int trump = fs.nextInt(), biden = fs.nextInt();
                int eVotes = fs.nextInt();
                if (trump > biden) trumpVotes += eVotes;
                else if (biden > trump) bidenVotes += eVotes;
            } else fs.next();   // skip non-integer token
        }
        System.out.println("Trump : " + trumpVotes + " votes");
        System.out.println("Biden : " + bidenVotes + " votes");
    }
}
```

Exercise 6: Working Hours

Given a file `hours.txt` with the following data content:

```
%ID %Name %Hours Worked Per Day
123 Kim   12.5  8.1 7.6 3.2
456 Eric   4.0 11.6 6.5 2.7 12
789 Stef   8.0  8.0 8.0 8.0  7.5
```

Write a JAVA application called `WorkingHours.java` that reads the data from the file `hours.txt`, processes the data on a token-by-token basis and computes the total number of hours worked by each person as well as the hourly working rate per day.

SAMPLE OUTPUT:

```
Kim  (ID#123) worked 31.4 hours (7.85 hrs/day)
Eric (ID#456) worked 36.8 hours (7.36 hrs/day)
Stef (ID#789) worked 39.5 hours (7.9  hrs/day)
```

Exercise 6: Solution

FILE FORMAT

```
123 Kim    12.5  8.1 7.6 3 2
456 Eric    4.0 11.6 6.5 2.1 12
```

```
import java.io.File; import java.util.Scanner;
public class WorkingHours {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner fs = new Scanner(new File("hours.txt"));
        while (fs.hasNext()) {
            int id = fs.nextInt(); String name = fs.next();
            double totHrs = 0.0; int days = 0;
            while (fs.hasNextDouble()) {
                totalHours += fs.nextDouble();
                day++;
            }
            System.out.println(name + " (ID#" + id + ") worked " +
                               totHrs + " hours (" +
        }
    }
}
```

!!! FLAWED SOLUTION !!!

PROBLEM

- Inner loop consuming ID of next worker.
- Care about line breaks (end of a record).

TO RESOLVE THE ISSUE

- Subdivide the content into different lines.
- Process each and every line individually on a token-by-token basis.

Line-based File Processing

- Useful methods for line-based file processing:

Method	Description
`nextLine()`	returns next entire line of input (from cursor to \n)
`hasNextLine()`	returns true if there are any more lines of input to read (always true for console input)

- **Sample Syntax:**

```
Scanner fs = new Scanner(new File("<f_name>.<ext>"));
Scanner ls; // Declare a line scanner;
while (fs.hasNextLine()) {
    String line = fs.nextLine();
    ls = new Scanner(line);
    while (ls.hasNext()) { ... Process the line ... }
}
```

Exercise 6: Corrected Solution

```
import java.io.*; import java.util.*;
public class WorkingHours {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner fs = new Scanner(new File("hours.txt")), ls;
        while (fs.hasNextLine()) {
            String line = fs.nextLine(); ls = new Scanner(line);
            int id = ls.nextInt(); String name = ls.next();
            double totHrs = 0.0; int days = 0;
            while (ls.hasNextDouble()) {
                totHrs += ls.nextDouble(); days++;
            }
            System.out.println(name + " (ID#" + id + ") worked " +
                               totHrs + " hours (" +
                               (totHrs/days) + " hours/day)");
        }
    }
}
```

Output To Files

AUB - CMPS 200 - Maurice J. KHABBAZ, Ph.D.

Monday, November 15, 2021

The `PrintStream` Class

- Is part of the `java.io` package.
- Defines an object for printing output to, for instance, a file.
- Has similar methods as `System.out` (e.g., `print`, `println`, etc).
- **Syntax:**

```
File f = new File("<f_name>.<ext>");
PrintStream <ps_name> = new PrintStream(f);
```

- **Example:**

```
PrintStream ps = new PrintStream(new File("out.txt"));
ps.println("Hello, file!");
ps.println("This is a second line of output.");
```

How `PrintStream` Operates

- If the given file:
  - **Does not exist** → **it is created**.
  - **Already exists** → **it is overwritten**.
- Printed output appears **in the file** (**not on the console/screen**):
  - To inspect the output, open the file with any available file editor.
- **Important Remark:**
  - Never open the same file for concurrently reading (`Scanner`) and writing (`PrintStream`).
  - Any attempt of writing to an opened file for reading will:
    - Erase the file's existing content.
    - Replace the old content with whatever new content is being written.

`System.out` and `PrintStream`

- The console output object, `System.out`, is a `PrintStream`:
  - A **reference** to `System.out` can be stored in a `PrintStream` variable.
  - Printing to that variable causes output to appear to the screen.
- It is also possible to pass `System.out` to a method:
  - As an input parameter of type `PrintStream`.
- Passing such parameters to methods allow these latter to:
  - Send output to the console/screen or a file.
- **Example:**

```
PrintStream out1 = System.out;
PrintStream out2 = new PrintStream(new File("data.txt"));
out1.println("Hello, console!");   // goes to console
out2.println("Hello, file!");      // goes to file
```

Exercise 7: Printing Output to a File

Modify our previous Hours program to use a `PrintStream` to send its output to the file `hours_out.txt`. The program must not produce any console output. However, the output file must have the output formatted as follows:

```
Kim (ID#123) worked 31.4 hours (7.85 hours/day)
Eric (ID#456) worked 36.8 hours (7.36 hours/day)
Stef (ID#789) worked 39.5 hours (7.9 hours/day)
```

Exercise 7: Solution

```
import java.io.File; import java.util.Scanner;
public class WorkingHours {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner fs = new Scanner(new File("hours.txt")), ls;
        PrintStream ps = new PrintStream(new File("hours_out.txt"));
        while (fs.hasNextLine()) {
            String fs = input.nextLine(); ls = new Scanner(line);
            int id = ls.nextInt(); String name = ls.next();
            double totHrs = 0.0; int days = 0;
            while (ls.hasNextDouble()) {
                totHrs += ls.nextDouble(); days++;
         }
         ps.println(name + " (ID#" + id + ") worked " +
                    totHrs + " hours (" + (totHrs / days) +
                    " hours/day)");
        }
    }
}
```

Prompting For a File Name

- It is possible to request from the user to interactively input the file to read:
  - This can be done using a Scanner object's `nextLine()` method.
  - The `next()` method does not work (possible existence of white spaces in file name).
- **Example:**

```
// prompt for file name input from user
Scanner k = new Scanner(System.in);
System.out.print("Enter a file name: ");
File f = new File(k.nextLine());        // Assume user enters sample.dat
while (!f.exists()) {                   // Test for existance of file
    System.out.print(filename + "not found! Enter valid file name: ");
    f = new File(k.nextLine());         // Take new filename as input
}
Scanner fs = new Scanner(f);
```

Mixture Tokens and Lines

- For the same `Scanner` object:
  - Using `nextLine()` in conjunction with token-based methods can misbehave.
- Consider the below file content:

```
23   3.14
  Joe   "Hello" world
          45.2  19
```

- One can be tricked into thinking that it is possible to:
  - Read `23` and `3.14` with `nextInt()` and/or `nextDouble()`
  - Then read `Joe "Hello" world` with `nextLine()`
- However:

```
System.out.println(input.nextInt());        // prints 23
System.out.println(input.nextDouble());     // prints 3.14
System.out.println(input.nextLine());       // No output
```

Why?

Example: Mixture Tokens and Lines

- Never read both tokens and lines using the same `Scanner` object:

```
23   3.14
  Joe   "Hello" world
          45.2  19
```

```
input.nextInt();        // reads: 23
```

```
23\t3.14\nJoe\t"Hello" world\n\t\t45.2  19\n
   ^
```

```
input.nextDouble();     // reads: 3.14
```

```
23\t3.14\nJoe\t"Hello" world\n\t\t45.2  19\n
         ^
```

```
input.nextLine();       // reads: "" Empty Line
```

```
23\t3.14\nJoe\t"Hello" world\n\t\t45.2  19\n
          ^
```

```
input.nextLine();       // reads: nJoe\t"Hello" world
```

```
23\t3.14\nJoe\t"Hello" world\n\t\t45.2  19\n
                      ^
```

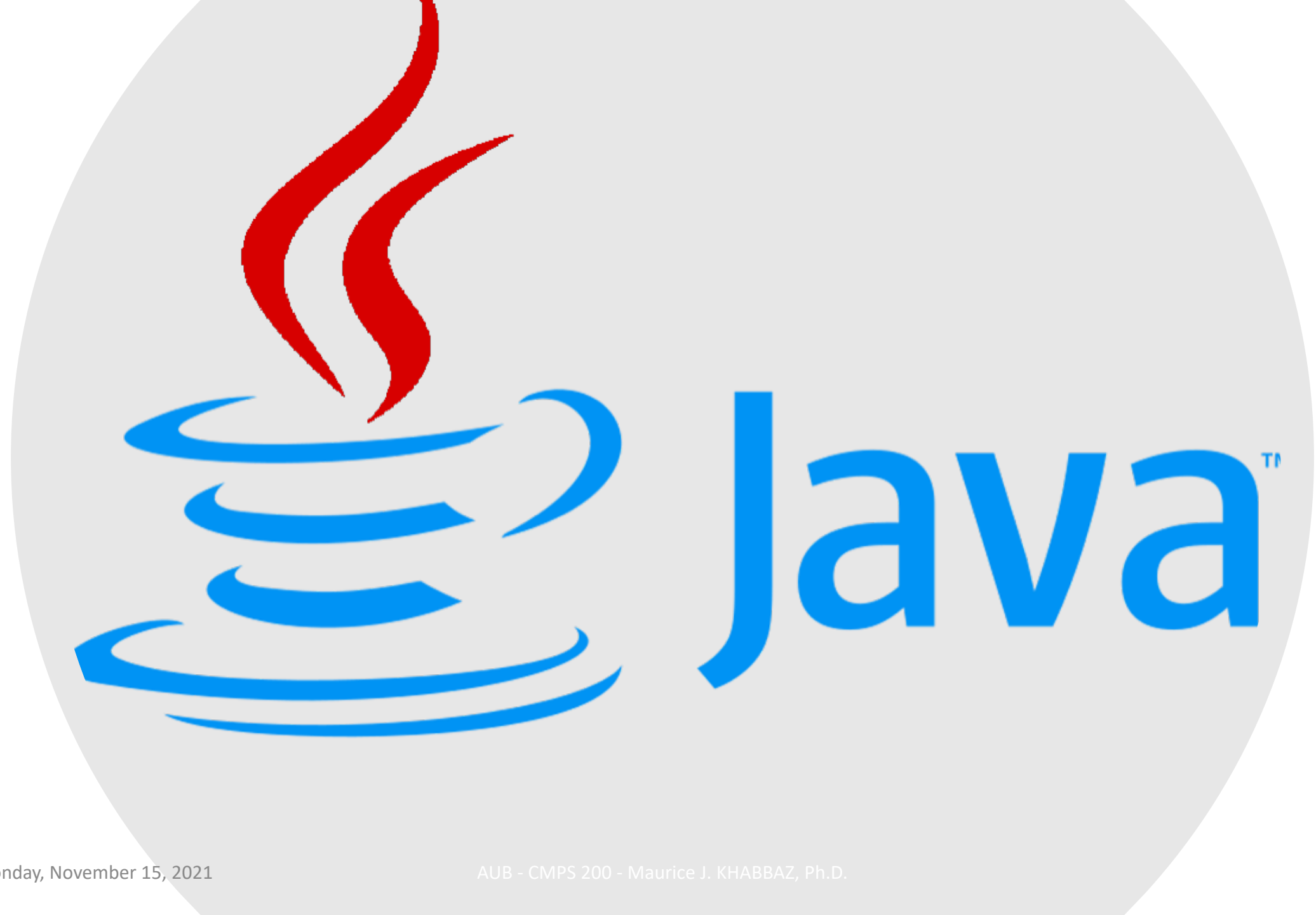
Another Example: Mixture Tokens and Lines

```
Scanner k = new Scanner(System.in);
System.out.print("Age: "); int age = k.nextInt();
System.out.print("Name: "); String name = k.nextLine();
System.out.println(name + " is " + age + " years old.");
```

- **Sample Execution (user input underlined):**

```
Age: 12
Name: Sideshow Bob
 is 12 years old.
```

- **Reason behind output discrepancy:**
  - Overall Input: `12\nSideshow Bob`
  - After `nextInt()`: `12\nSideshow Bob`
    `^`
  - After `nextLine()`: `12\nSideshow Bob`
    `^`
- **Remark: Need another `nextLine()` to read `name`.**


Java