Advanced Programming Techniques in Java

COSI 12B

Class objectives

- Files I/O (Chapter 6)
- Arrays (Chapter 7)

Review: Compiler error with files

```
import java.io.*;       // for File
import java.util.*;     // for 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 program fails to compile with the following 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

- An **exception** is an error that occurs at runtime as a result of some type of "exceptional" circumstance
  - Dividing an integer by 0
  - Calling substring on a `String` and passing too large an index
  - Trying to read the wrong type of value from a Scanner
  - Trying to read a file that does not exist

```
StringIndexOutOfBoundsException
IllegalArgumentException
```

Review: Exceptions

- *Checked* exceptions
  - normally not due to programmer error
  - generally, beyond the control of the programmer
  - all I/O errors are checked exceptions
  - eg. FileNotFoundException
- *Unchecked exceptions*
  - programmer error (try to prevent them with defensive programming)
  - a serious external condition that is unrecoverable
  - eg. ArrayIndexOutOfBoundsException

Review: Exceptions

- When using a `Scanner` to process a file, we can get a `FileNotFoundException`:
  - If the file that we specify isn't there
  - If the file is inaccessible for some reason

- We say that a program with an error "throws" an exception
- It is also possible to "catch" (handle or fix) an exception

- The compiler checks that we either
  - **Declare that we don't handle it**
  - **Handle it (`try/catch`)**

- We do this by adding a throws clause

Token-based vs. line-based processing

- Token-based: The practice of processing input token by token (i.e., one word at a time or one number at a time)
- Line-based: The practice of processing input line by line (i.e., reading in entire lines of input at the time)

Input token

- A token is unit of user input, separated by whitespace
- The `Scanner` methods don't necessarily read an entire line of output
- If the input file contains the following:

```
23 3.12
"Iraklis"
```

- The `Scanner` can interpret the tokens as the following types:

Token	Type(s)
`23`	`int, double, String`
`3.12`	`double, String`
`"Iraklis"`	`String`

Files and input cursor

- Consider a file `weather.txt` that contains this text

```
16.2 23.5
19.1 7.4 22.8

18.5 -1.8 14.9
```

- A `Scanner` views all input as a stream of characters

```
16.2 23.5\n19.1 7.4 22.8\n\n18.5 -1.8 14.9\n
^
```

Consuming tokens

- **Consuming input** means reading input and advancing the cursor
- Calling `nextInt` etc. moves the cursor past the current token

```
16.2 23.5\n19.1 7.4 22.8\n\n18.5 -1.8 14.9\n
^

double d = input.nextDouble();     // 16.2
16.2 23.5\n19.1 7.4 22.8\n\n18.5 -1.8 14.9\n
       ^

16.2 23.5\n19.1 7.4 22.8\n\n18.5 -1.8 14.9\n
                                            ^
```

- If you attempted to call `nextDouble` again, it would throw a `NoSuchElementException`

`Scanner` tests for valid input

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

- These methods of the `Scanner` do not consume input, they just give information about what the next token will be
  - Useful to see what input is coming, and to avoid crashes
- They can be used with a console `Scanner`, as well

Files input: Question 1

- Consider a file `weather.txt` that contains this text

```
16.2 23.5
19.1 7.4 22.8

18.5 -1.8 14.9
```

- Write a program that prints the change in temperature between each pair of neighboring days

```
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
```

Files input: Answer 1

```
// Displays changes in temperature from data in an input file

import java.io.*;     // for File
import java.util.*;   // for Scanner
public class Temperatures {
  public static void main(String[] args) throws FileNotFoundException {

      Scanner input = new Scanner(new File("weather.txt"));

      double prev = input.nextDouble();
      while (input.hasNextDouble()) {
         double next = input.nextDouble();
         System.out.println(prev + " to " + next + ", change = " + (next - prev));
         prev = next;
      }
   }
}
```

Files input: Question 2

- Modify the temperature program to handle files that contain non-numeric tokens (by skipping them)

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

18.5 -1.8 14.9
16.1
```

- You may assume that the file begins with a real number

Files input: Answer 2

```java
// Displays changes in temperature from data in an input file

import java.io.*;     // for File
import java.util.*;   // for Scanner
public class Temperatures2 {
    public static void main(String[] args) throws FileNotFoundException {
        Scanner input = new Scanner(new File("weather.txt"));
        double prev = input.nextDouble();
        while (input.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println(prev + " to " + next + ", change = " + (next - prev));
                prev = next;
            } else {
                input.next();   // throw away unwanted token
            }
        }
    }
}
```

Line-based `Scanner`

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)

```
Scanner input = new Scanner(new File("file name"));
while (input.hasNextLine()) {
    String line = input.nextLine();
    processLine(line);
}
```

`Scanner` **on strings**

- So far we have seen that you can pass to `Scanner` the object `System.in` and the object `File`
- We can also pass the object `String`

Scanner on strings

- A Scanner can tokenize the content of a String

Syntax:

```
Scanner <name> = new Scanner(<String>);
```

Example:

```
String text = "15  3.2 hello   9 27.5";
Scanner scan = new Scanner(text);

int num = scan.nextInt();   // 15
double num2 = scan.nextDouble();  //3.2
String word = scan.next(); //hello
```

Mixing lines and tokens

Input file `input.txt`:	Output to console:

`The quick brown fox jumps over`
`the lazy dog.`
	`Line has 6 words`
	`Line has 3 words`

```
// Counts the words on each line of a file
Scanner input = new Scanner(new File("input.txt"));
while (input.hasNextLine()) {
        String line = input.nextLine();
        Scanner lineScan = new Scanner(line);

        // process the contents of this line
        int count = 0;
        while (lineScan.hasNext()) {
                String word = lineScan.next();
                count++;
        }
        System.out.println("Line has " + count + " words");
}
```

File output

- So far we have sent the output of a program to the console window
  - `System.out.print`
  - `System.out.println`
- You can write output to a file:

Syntax

```
PrintStream <name> = new PrintStream (new File ("results.txt");
```

Example

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

Details about `PrintStream`

Syntax

```
PrintStream <name> = new PrintStream (new File ("results.txt");
```

- If the given file does not exist, it is created
- If the given file already exists, it is overwritten

- The output you print appears in a file, not on the console
  - You will have to open the file with an editor to see it

- Do not open the same file for both reading (`Scanner`) and writing (`PrintStream`) at the same time
  - You will overwrite your input file with an empty file

Details about `PrintStream`

Syntax

```
PrintStream <name> = new PrintStream (new File ("results.txt");
```

- This line of code can generate an exception if Java is unable to create the file
  - You might not have permission to write to the directory
  - You might be locked because another file is using it
- To handle the exception, you need to include the `throws` clause in whatever method contains this line of code or surround it with a `try/catch`.

System.out and PrintStream

- The console output object System.out, is a PrintStream

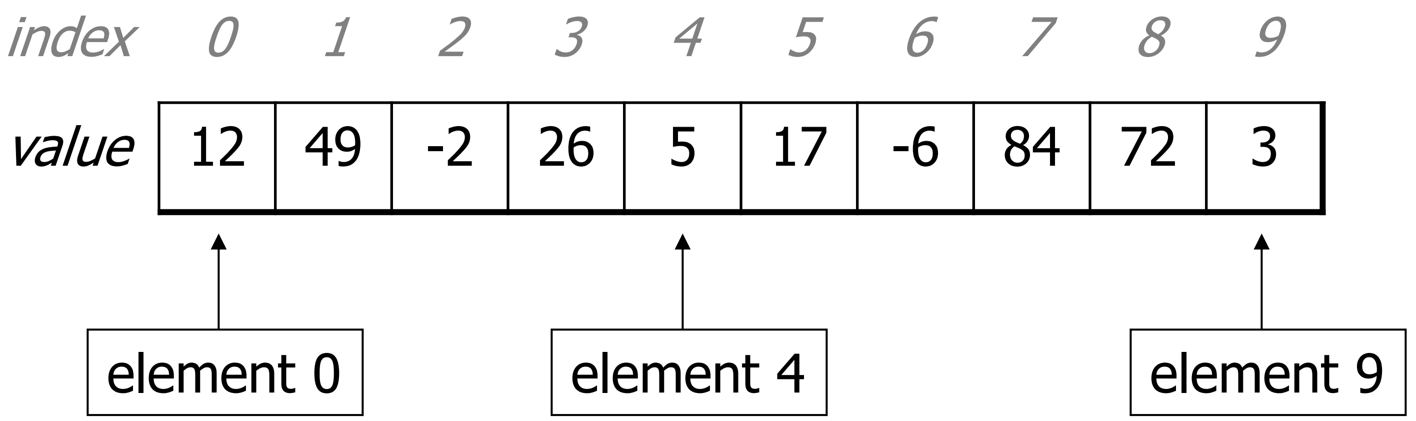
```
    PrintStream out1 = System.out;
    PrintStream out2 = new PrintStream(new File("data.txt"));
    out1.println("Hello, console!");   // goes to console
    out2.println("Hello, file!");      // goes to file
```

- A reference to it can be stored in a PrintStream variable
- You can pass System.out to a method as a PrintStream

Arrays

Arrays

- An array is a collection (object) of data values (or elements) of the same type
- An array can be thought as a sequence of boxes



index *0* *1* *2* *3* *4* *5* *6* *7* *8* *9*
value 12 49 -2 26 5 17 -6 84 72 3
element 0
element 4
element 9

- Each box contains one of the data values in the collection
- Each element has a numeric index. The first element has index of 0

Declaring and creating an array

- We often declare and create an array in the same statement

```
<type>[] <array_name> = new <type>[<length>];
```

type of the individual elements

number of elements in the array

```
int[] A = new int[10];
```

The length of an array

- The length of an array is the number of elements in the array
- The length of an array can be obtained as follows: `<arrayName>.length`

- Example: `A.length`

- NOTE: length is not a method

`data.length()` won't work

Auto initialization

- When you create an array in this way: `int[] A = new int[10];` the elements are initialized to 0

	index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
A	*value*	0	0	0	0	0	0	0	0	0	0

- Each element initially gets a "zero-equivalent" value

Type	**Default value**
`int`	`0`
`double`	`0.0`
`boolean`	`false`
`String`	`null`

Accessing elements in an array

- To access the elements in an array, we use the expression: `<arrayName>[<index>]`

	index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
A	*value*	27	0	0	-6	0	0	0	0	0	0

`A[0]`	access the first element
`A[3]`	access the fourth element

Modifying elements in an array

A

index	0	1	2	3	4	5	6	7	8	9
value	0	0	0	0	0	0	0	0	0	0

- To modify an elements in an array, we use the expression:

```
<arrayName>[<index>] = value;
```

```
A[0] = 27;

A[3] = -6;
```

A

index	0	1	2	3	4	5	6	7	8	9
value	27	0	0	-6	0	0	0	0	0	0

Accessing elements in an array

- Legal index values: integers from `0` to `<array_name>.length-1`
- Reading or writing any index outside this range will throw an

  `ArrayIndexOutOfBoundsException`

```
int[] A = new int[10];
System.out.println(A[0]);          // okay
System.out.println(A[9]);          // okay
System.out.println(A[-1]);         // exception
System.out.println(A[10]);         // exception
```

A

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	0	0	0	0	0	0	0	0	0

Accessing elements in an array

- The index can be any integer expression: `int lastData = A[A.length - 1];`

- We can operate on an array element in the same way that we operate on any other variable of that type
- Example: Applying a 10% late penalty to the data at index i

```
A[i] = (int)(A[i]* 0.9);
```

Another way to create an array

- If we know that we want an array to contain specific values, we can specify them when create the array `int[] data = {7, 8, 9, 6, 10, 7, 9, 5};`
- This list of values is known as an **initialization list**
- We don’t use the `new` operator in this case
- We don’t specify the length of the array (it is determined from the number of values in the initialization list)

```
double[] heights = {65.2, 72.0, 70.6, 67.9};
```

```
boolean[] isPassing = {true, true, false, true};
```

Arrays of other types

```
double[] results = new double[5];
results[2] = 3.4;
results[4] = -0.5;
```

results

index	*0*	*1*	*2*	*3*	*4*
value	0.0	0.0	**3.4**	0.0	**-0.5**

```
boolean[] test = new boolean[6];
test[3] = true;
```

test

index	*0*	*1*	*2*	*3*	*4*	*5*
value	false	false	false	**true**	false	false

Traversing Arrays

- Often, we will want to do something like walk down an array and do something to each cell in the array. We use a for loop:

```
int[] primes = {2, 3, 5, 7, 11, 13, 17};

for(int i = 0; i < primes.length; i++){
      System.out.println( primes[i]) );
}
```

Arrays and static methods

- **Method declaration**
- Syntax:

```
public static type methodName(type[] arrayName){
```

- Write a method that returns the average of the given array of numbers

Arrays and static methods

- **Method declaration**
- Syntax: `public static type methodName(type[] arrayName){`

- Write a method that returns the average of the given array of numbers

```
public static double average(int[] numbers) {
    int sum = 0;
    for (int i = 0; i < numbers.length; i++) {
       sum += numbers[i];
    }
    return (double) sum / numbers.length;
}
```

- You don’t specify the array’s length (but you can examine it)

Arrays and static methods

- **Method call**
- <u>Syntax</u>: `methodName(arrayName);`

- Write a method that returns the average of the given array of numbers

```
public class MyProgram {
      public static void main(String[] args) {

          int[] iq = {126, 84, 149, 167, 95};
          double avg = average(iq);
          System.out.println("Average IQ = " + avg);
      }
      ...
```

- Notice that you don’t write the `[]` when passing the array

Arrays and static methods

- **Return an array –** method declaration
- Syntax:

```
public static type[] methodName(parameters) {
```

- Write a method that returns an array with two copies of each value

```
[1, 4, 0, 7] -> [1, 1, 4, 4, 0, 0, 7, 7]
```

Arrays and static methods

- **Return an array –** method declaration
- Syntax:

```
public static type[] methodName(parameters) {
```

- Write a method that returns an array with two copies of each value

```
[1, 4, 0, 7] -> [1, 1, 4, 4, 0, 0, 7, 7]
```

```
public static int[] twoCopies(int[] numbers) {
      int[] result = new int[2 * numbers.length];
      for (int i = 0; i < numbers.length; i++) {
          result[2 * i]     = numbers[i];
          result[2 * i + 1] = numbers[i];
      }
      return result;
  }
```

Arrays and static methods

- **Return an array –** method call
- Syntax:

```
type[] arrayName =  methodName(parameters);
```

- Write a method that returns an array with two copies of each value

```
[1, 4, 0, 7] -> [1, 1, 4, 4, 0, 0, 7, 7]
```

```
public class MyProgram {
      public static void main(String[] args) {
          int[] iq = {126, 84, 149, 167, 95};
          int[] out = twoCopies(iq);
                    System.out.println(Arrays.toString(out));
      }
      ...
```

Limitations of arrays

- You cannot resize an existing array

```
int[] A = new int[4];
A.length = 10;          // error
```

- An array does not know how to print itself

```
int[] A1 = {42, -7, 1, 15};
System.out.println(A1);
```

- You cannot compare arrays with `==` or `.equals` for Strings)

```
int[] A1 = {42, -7, 1, 15};
int[] A2 = {42, -7, 1, 15};
if (A1 == A2) {  ... }          // false!
if (A1.equals(A2)) {  ... }     // false!
```

Limitations of arrays

```
public static void main(String[] args) {
        int[] A = {126, 167, 95};
        int[] B = A;
        int[] C = {126, 167, 95};

        System.out.println("A location = " + A);
        System.out.println("B location = " + B);
        System.out.println("C location = " + C);

        System.out.println(Arrays.toString(A));
        System.out.println(Arrays.toString(B));
        System.out.println(Arrays.toString(C));
}
```