



**SRI AKILANDESWARI WOMEN'S COLLEGE, WANDIWASH**

# **File Input/Output and Exceptions**

**Class : II B.C.A**

**Mrs.R.SAIKUMARI**

**Assistant Professor**

**Department of Computer Applications**

**SWAMY ABEDHANADHA EDUCATIONAL TRUST, WANDIWASH**

# Chapter Topics

We will cover the following topics in this order:

- 11.1.5 The Scanner Class
- 11.1.6 Serialized Object I/O
- 2.4.1 Basic Output and Formatted Output
- 11.2 Files
- 3.7 Exceptions and try .. catch
- 8.3 Exceptions and try .. catch

# Review Text Files - Reading

To read in from a file using Scanner, we need to tell the Scanner where to read in from.

1. import **java.io.\***;
2. Create a **File** object and provide it the name of the file
3. Pass the **File** object to the **Scanner**
4. Use the **hasNext()** method to determine if there is still data to read.
5. Use the Scanner's **.next** methods to read the data.
6. When done, **close** the file.

```
File file = new File("animals.txt");  
Scanner input = new Scanner(file);  
while(input.hasNext())  
{  
    String animal = input.nextLine();  
    System.out.println(animal);  
}  
in.close();
```

# Review Text Files - Writing

To write to a file, use the **PrintWriter** class.

1. import **java.io.\***;
2. Create a **PrintWriter** object that takes the file name as input
3. Use the `.print()`, `.println()` and `.printf()` methods to write.
4. Close the **PrintWriter**. Failure to close the **PrintWriter** will not save the contents written.

Reminder: **PrintWriter** will create the file if it does not exist and will overwrite the contents if it had content.

```
PrintWriter myFile = new PrintWriter("essay.txt");  
myFile.println("I love Java.");  
myFile.close();
```

# Scanner `.next()` to Read a Single Word

- In the past, we have been using `.nextLine()` to read an entire line from a file. What if we wanted to read just a single String? Use **`.next()`**. It will read until a white space is encountered (could be an enter, a tab, or a space).

```
while (in.hasNext())
{
    String s = in.next();
}
```

# Scanner .useDelimiter(...)

- In Java, the default delimiter for the Scanner is white space. This delimiter tells the Scanner when it should stop looking for the next item to read.
- The Scanner has a method **.useDelimiter** (String pattern) which can set the scanner's delimiting pattern to a pattern constructed from the input String.

## Example:

Changing the delimiter to the empty String instructs the Scanner to stop reading after every character.

**.next()** here will give back one character. So this code will read one letter at time.

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

while (in.hasNext())
{
    char c = in.next().charAt(0);
    // do something with c
}
```

# Scanner .useDelimiter() and Patterns

- You can use patterns with the .useDelimiter() method.

[^abc]	Any character except a, b, or c ( <b>negation</b> )
[a-zA-Z]	a through z or A through Z, inclusive ( <b>range</b> )
X+	X repeated <b>one or more</b> times

- For example, if you want to read only letters and skip all other characters (including numbers and special characters), you can use `.useDelimiter("[^a-zA-Z]+")`

This is saying the delimiter is anything that is not (^) the letters a-z or A-Z repeated (+)

```
Scanner in = new  
Scanner("input.txt");  
in.useDelimiter("[^a-zA-Z]+");
```

# Parsing a String using Scanner

- A Scanner can take a String as input to parse it.

```
String line = "Ibtsam M Mahfouz";  
Scanner lineScanner = new Scanner(line);  
String first = lineScanner.next();  
String middle = lineScanner.next();  
String last = lineScanner.next();
```

- Example with numbers:

```
String streetNum = "123 Main Street";  
Scanner scanner = new Scanner(streetNum);  
int number = scanner.nextInt();  
String street = scanner.next() + " " + scanner.next();
```



# Scanner's `.hasNext<Type>` Methods

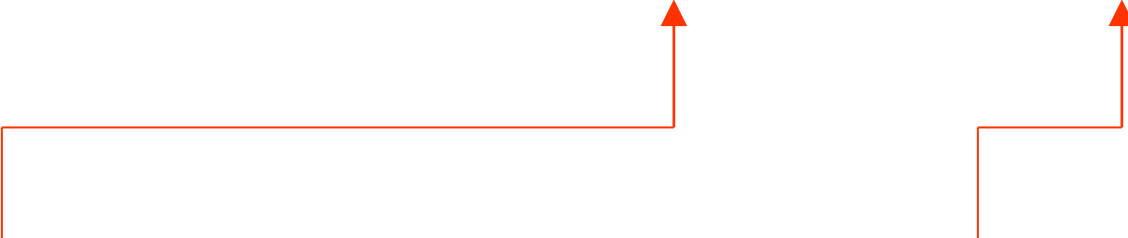
- **`hasNext()`** — returns a boolean value that is true if there is at least one more token in the input source.
- **`hasNextInt()`, `hasNextDouble()`**, and so on—return a boolean value that is true if there is at least one more token in the input source and that token represents a value of the requested type.
- **`hasNextLine()`** — returns a boolean value that is true if there is at least one more line in the input source.

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

# Review System.out.printf

- You can use the `System.out.printf` method to perform formatted console output.

```
System.out.printf(FormatString, ArgList);
```



***FormatString*** is a string that contains text and/or special formatting specifiers.

***ArgList*** is optional. It is a list of additional arguments that will be formatted according to the format specifiers listed in the format string.

# Formatting Output – System.out.printf

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:

```
int hours = 40;
```

```
System.out.printf("I worked %d hrs.", hours);
```

```
double pay = 874.12;
```

```
System.out.printf("Your pay is %f.\n", pay);
```

```
String name = "Ibtsam";
```

```
System.out.printf("My name is %s.\n", name);
```

# Formatting Output – Additional Options

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
S	Convert letters to uppercase	1.23E+1

```
int amount = -40;
// will print (40)
System.out.printf("%(d hrs.", amount);
// will print (040) - () are counted in the 5.
System.out.printf("%(05f).\n", amount);
String name = "Ibtsam";
System.out.printf("My name is %S.\n", name);
```

# Formatting Output – Additional Options

```
String item = "computer";  
double price = 499.99;  
System.out.printf("%-10s%10.2f", item + ":",  
price);
```

- %-10s
  - Formats a left-justified string
  - Padded with spaces so it becomes 10 characters wide
- %10.2f
  - Formats a floating-point number
  - 2 numbers after the decimal
  - The field that is ten characters wide.
  - Right justified

# CW Part-4-1: Input & Output

- Download the Student.java file attached to the CW on Blackboard and use its main template.
- Create a class called Student which stores
  - the name of the student
  - the grade of the student
- Write a main method that asks the user for the name of the input file and the name of the output file. Main should open the input file for reading. It should read in the first and last name of each student into the Student's *name* field. It should read the grade into the *grade* field.
- Calculate the average of all students' grades.
- Open the output file, for writing, and print all the students' names on one line and the average on the next line.
  - Average should only have 1 digit after the decimal
  - "Average" should be left justified in a field of 15 characters. The average # should be right justified in a field of 10 spaces.

Compile and test your code in NetBeans and then on Hackerrank at

<https://www.hackerrank.com/csc128-part-4-classwork> then choose CSC128-Classwork-Part-4-1

Submit your .java file and a screenshot of passing all test cases on Hackerrank.

Sample  
Input

```
Minnie Mouse 98.7
Bob Builder 65.8
Mickey Mouse 95.1
Popeye SailorMan 78.6
```

Output

```
Minnie Mouse, Bob Builder, Mickey Mouse, Popeye SailorMan
Average:      84.6
```

# Exceptions

- An **exception** is an exception to the normal flow of control in the program. The term is used in preference to “error” because in some cases, an exception might not be considered to be an error at all.
- Exceptions in Java are represented as objects of type **Exception**.
- If exceptions are not handled, the program will crash and the **default exception handler** will print the stack trace showing where the exception was thrown in the code.

# Exceptions – using try/catch

- When an exception occurs, we say that the exception is **thrown**. For example, we say that `Integer.parseInt(str)` throws an exception of type `NumberFormatException` when the value of `str` is illegal.
- When an exception is thrown, it is possible to **catch** the exception and prevent it from crashing the program. This is done with a try/catch statement.

```
try  
{  
    try block statements  
}
```

The program executes the statements in the try block. If no exception occurs the program skips the catch part and proceeds with the rest of the program.

```
catch (ExceptionType ParameterName)  
{  
    catch block statements  
}
```

If an exception occurs during the execution of the try block, the program immediately jumps from the point where the exception occurs to the catch part and executes the catch block statements, skipping any remaining statements in the try block.



# Exception Types – Examples

- When specifying the catch clause, you need to specify the type of exception to catch based on the code in the try block.

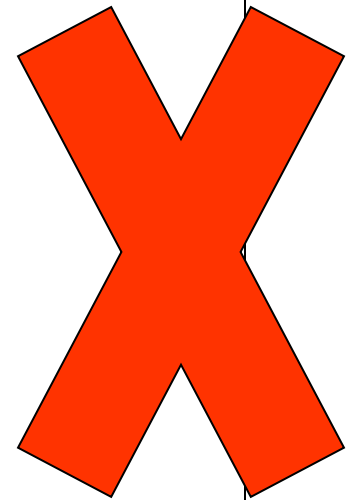
```
double x;  
String str = scanner.next();  
try {  
    x = Double.parseDouble(str);  
    System.out.println( "The number is " + x );  
}  
catch ( NumberFormatException e )  
{  
    System.out.println( str + "is not a legal number." );  
}
```

```
String filename = scanner.nextLine();  
try {  
    File f = new File(filename);  
    Scanner inFile = new Scanner(f);  
}  
catch ( IOException e )  
{  
    System.out.println("Failed to open " + filename );  
}
```

# Exception Types – Incompatible Types

- If the exception being thrown is not the one caught, the code will still crash!
- This code will throw a `NumberFormatException`. Since this code is not catching the correct exception, the code will still crash.

```
double x;  
String str = scanner.next();  
try {  
    x = Double.parseDouble(str);  
    System.out.println( "The number is " + x );  
}  
catch ( IOException e )  
{  
    System.out.println( str + "is not a legal number." );  
}
```



# Once an Exception is Caught

The exception object in the catch has useful methods:

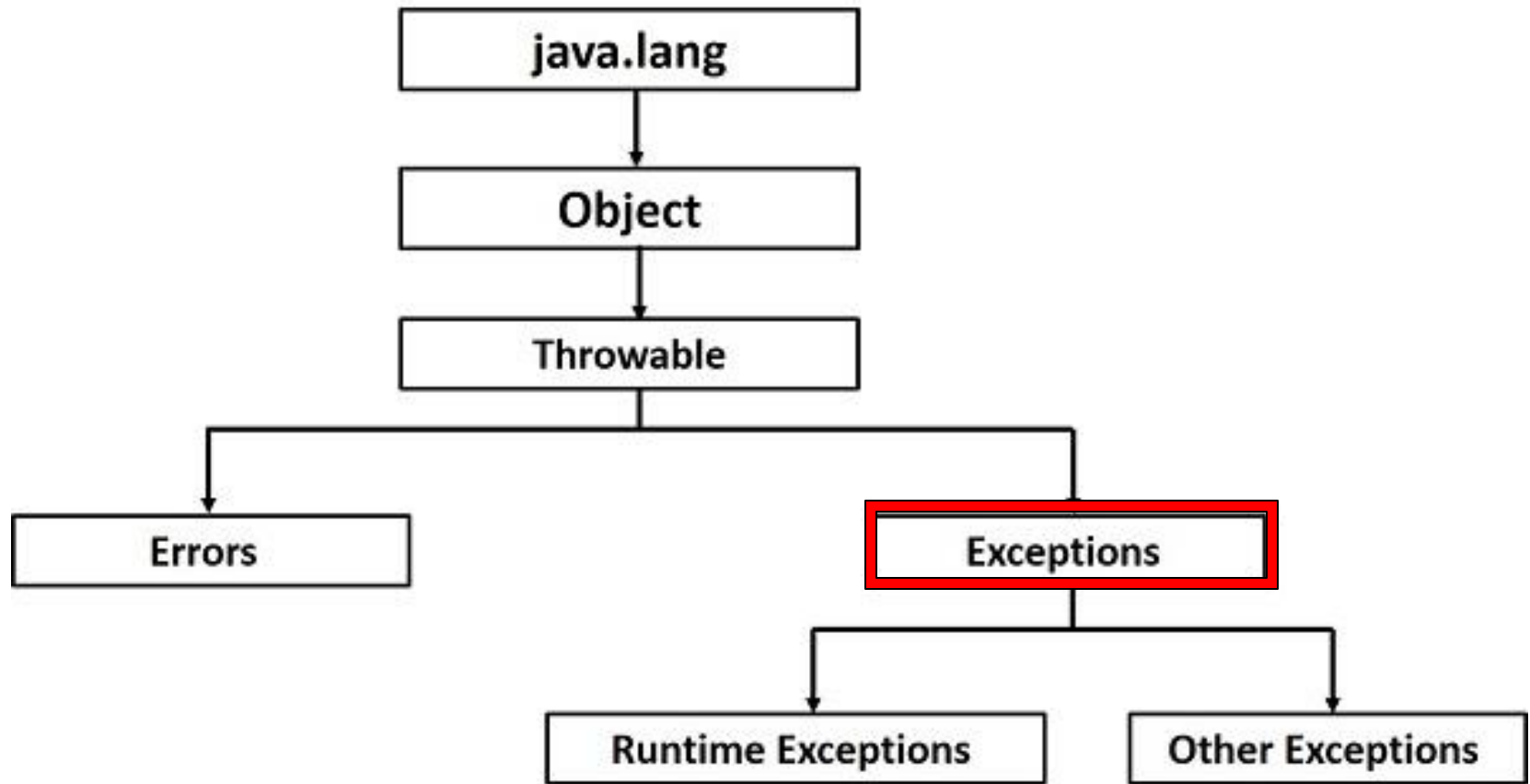
- **getMessage()**: returns the default error message for the exception.
- **printStackTrace()**: writes a stack trace to standard output that tells which methods were active when the exception occurred. A stack trace can be very useful when you are trying to determine the cause of the problem. (Note that if an exception is not caught by the program, then the default response to the exception prints the stack trace to standard output.)

# Catching Multiple Exceptions

```
try
{
    String filename = "d:\\test.txt";
    Scanner in = new Scanner(new File(filename));
    int sum = 0;
    while (in.hasNext())
    {
        String input = in.next();
        int value = Integer.parseInt(input);
        sum = sum + value;
    }
    System.out.println("Sum: " + sum);
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    exception.printStackTrace();
    System.out.println(exception.getMessage());
}
```

# Exception Classes

An **Exception** is an **Object**.



# Exception vs. Error

- Most of the subclasses of **Error** represent serious problems that you as a programmer should not try to handle. They usually represent problems that occurred within the Java virtual machine.
- We will handle exceptions that are subclasses of **Exception**.

# Catching Multiple Exceptions

```
try
{
    ...
}
catch (IOException exception)
{
    exception.printStackTrace();
}
catch (NumberFormatException exception)
{
    exception.printStackTrace();
    System.out.println(exception.getMessage());
}
```

```
try
{
    ...
}
catch (Exception exception)
{
    exception.printStackTrace();
}
```

This code will catch all **Exceptions**.

```
try
{
    ...
}
catch (IOException | NumberFormatException exception)
{
    exception.printStackTrace();
}
```

This code will only catch **IOException** and **NumberFormatException**

# Finally Clause

- The try statement can have an optional **finally** clause after the catch clauses.
- The **finally** clause is guaranteed to be executed as the last step in the execution of the try statement, whether or not any exception occurs and whether or not any exceptions that do occur are caught and handled.
- The **finally** clause is meant for doing essential cleanup that under no circumstances should be omitted. One example of this type of cleanup is closing a file or a network connection.

```
try
{
    try block statements...
}
catch (ExceptionType varName)
{
    catch block statements...
}
finally
{
    finally block statements...
}
```



# Propagating Exceptions

- All exceptions thrown must be handled in a Java program.
- If your program does not handle it, the default exception handler will handle it by halting the execution of the program and printing the stack trace.
- If an exception happens inside a method and this method does not have an exception handler, control is passed up the call stack to the calling method. It will propagate back until it reaches the *main* method.
- The *main* method will either handle it or the default exception handler will handle it and print the stack trace.

```
public static void doWork() throws IOException
{
    String filename = "e:\\test1.txt";
    Scanner in = new Scanner(new File(filename));
    int sum = 0;
    while (in.hasNext())
    {
        String input = in.next();
        int value = Integer.parseInt(input);
        sum = sum + value;
    }
    System.out.println("Sum: " + sum);
}
```

```
public static void main(String[] args)
{
    try
    {
        doWork();
    }
    catch(Exception exception)
    {
        exception.printStackTrace();
        System.out.println(exception.getMessage());
    }
}
```

# Checked and Unchecked Exceptions

Exceptions can either be :

- **Unchecked:** These are the exceptions that are derived from the **Error** class or the **RuntimeException** class and you do not need to handle in your code.
- **Checked:** These are exceptions that require mandatory handling. These come from methods who have declared that they throw exceptions by adding **throws** *<ExceptionType>* in their method header.

```
// This method will not compile!  
public void displayFile(String name)  
{  
    // Open the file.  
    File file = new File(name);  
    Scanner in = new Scanner(file);  
    ...  
}
```

**To fix it:**

```
public void displayFile(String name)  
    throws FileNotFoundException
```

# CW Part-4-2: Exceptions

- Download the Student.java file attached to the CW on Blackboard and use its main template.
- Modify the code written for CW Part-4-1 to detect if any line does not contain the three fields: first name, last name and grade or if the grade is not parse-able to a float. Catch any exceptions thrown, print the message from the exception and a meaningful error message along with the problem line.
- The exception/error message should be for the line that is not correct only. Main should continue parsing the remaining lines.

Compile and test your code in NetBeans and then on Hackerrank at

- <https://www.hackerrank.com/csc128-part-4-classwork> then choose CSC128-Classwork-Part-4-2

Submit your .java file and a screenshot of passing all test cases on Hackerrank.

# Programming with Exceptions

You can write code that throws an exception if it encounters an error.

You can either:

- use the built in **Exception** class to throw an exception with an appropriate error message

```
throw new Exception("Unable to withdraw amount");
```

- create a new exception class that extends **Exception** and add the appropriate constructors.

```
public class NegativeBalanceException extends Exception
```

```
{
```

```
/**
```

```
    This constructor uses a generic error message.
```

```
*/
```

```
public NegativeBalanceException()
```

```
{
```

```
    super("Error: Negative balance");
```

```
}
```

```
/**
```

```
    This constructor specifies the balance & amount causing it to become negative
```

```
*/
```

```
public NegativeBalanceException(double balance, double amount)
```

```
{
```

```
    super("Error: Balance is " + balance +  
        ". Deducting " + amount + " will make it negative");
```

```
}
```

```
}
```

```
/**
 * @param startBalance
 * @throws NegativeBalanceException
 */
public BankAccount(double startBalance) throws NegativeBalanceException
{
    if (startBalance < 0)
        throw new NegativeBalanceException();

    balance = startBalance;
}
```

```
/**
 * @param amount
 * @throws NegativeBalanceException
 */
public void withdraw(double amount) throws NegativeBalanceException
{
    if(balance - amount < 0)
        throw new NegativeBalanceException(balance, amount);
}
```

# Nested Try/Catch

```
String[] array = { null };
for(String s : array)
{
    try
    {
        System.out.println(Integer.parseInt(s));
    }
    catch(Exception e)
    {
        try
        {
            System.out.println("invalid value " + s);
            FileWriter f = new FileWriter("z:\\errors.txt", true);
            PrintWriter pw = new PrintWriter(f);
            pw.println("invalid value " + s);
            pw.close();
        }
        catch(Exception ee)
        {
            System.out.println(ee.getMessage());
        }
    }
}
```

1<sup>st</sup> try/catch  
responsible  
for catching  
Strings that  
are not  
integers.

2<sup>nd</sup> try/catch  
responsible for  
printing an error  
message to a  
file. But opening  
the file will  
cause another  
exception if the  
path is wrong.



# Javadocs: @exception Tag

- Use the **@exception** tag in the comments to document that a method throws an exception.
- It should appear after the description of the method.

**@exception** ExceptionType Description

# CW Part-4-3: Exception Class

- Download the Student.java file attached to the CW on Blackboard and use its main template.
- Modify the Student constructor written for CW 11-2 to detect if any line contains a grade less than 0 or greater than 100. If it does, it should throw an exception.
- Write your own exception class to do the above task. It could have several constructors to handle the different error cases.
- Modify main to handle the new exception thrown from the Student constructor. It should continue processing the valid lines.

Compile and test your code in NetBeans and then on Hackerrank at <https://www.hackerrank.com/csc128-part-4-classwork> then choose CSC128-Classwork-Part-4-3

Submit your .java file and a screenshot of passing all test cases on Hackerrank.

# Homework (50 Points)

1. Create an exception class called **InvalidStringException** which can handle two types of errors:
  - A String that is too long. The program detecting the exception should pass the length of the String and the maximum allowed length
  - A String that contains digit characters.
2. Write a class ValidStrings which stores an ArrayList of Strings.
  - Its constructor should take the maximum length allowed.
  - It should have an add method which takes a String and adds it to the ArrayList if it is valid (length and does not contain digits). If the String is invalid, it should throw an appropriate InvalidStringException.
3. Write a main method that:
  - Asks the user for the name of the input file.
  - Asks the user for the maximum string length allowed.
  - Creates a ValidStrings object.
  - Opens the input file for reading. It will read one word at a time (all words will be on the same line) and tries to add it to the ValidStrings object. If it fails, it should print the exception message returned and continue processing the rest of the file.
  - Print the Strings in the ArrayList in the ValidStrings object separated by spaces.

Compile and test your code in NetBeans and then on Hackerrank at

<https://www.hackerrank.com/contests/csc128-programmingassignments> then choose CSC128-Part-4-PA

Submit your .java file and a screenshot of passing all test cases on Hackerrank.

# Acknowledgment

Attribution-NonCommercial-  
ShareAlike 4.0 International



"Java II – Part 4 – Input, Output and Exceptions" by Ibtisam Mahfouz, [Manchester Community College](#) is licensed under [CC BY-NC-SA 4.0](#) / A derivative from the [original work](#)