IO BASIC
EXCEPTION HANDLING

Instructor:

Table of contents

Learning Approach

Strongly suggested for a better learning and understanding of this course:

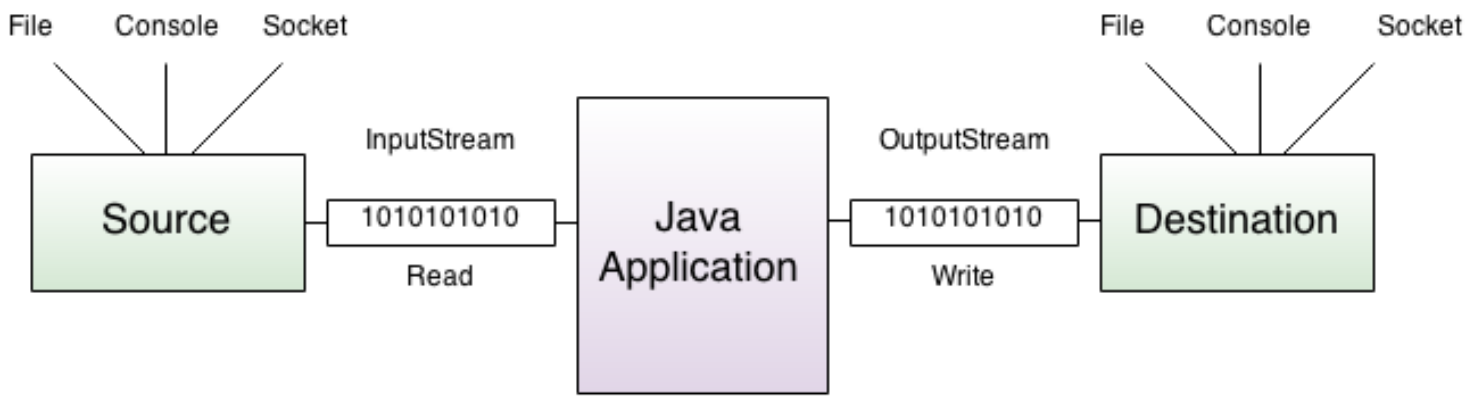
- Noting down the ***key concepts*** in the class
- ***Analyze*** all the examples / code snippets provided
- Study and understand the ***self study topics***
- Completion and ***submission*** of all the assignments, on time
- Completion of the ***self review*** questions in the lab guide
- ***Study*** and understand all the artifacts
- ***Completion*** of the project on time inclusive of individual and group activities

Section 1

JAVA IO BASIC

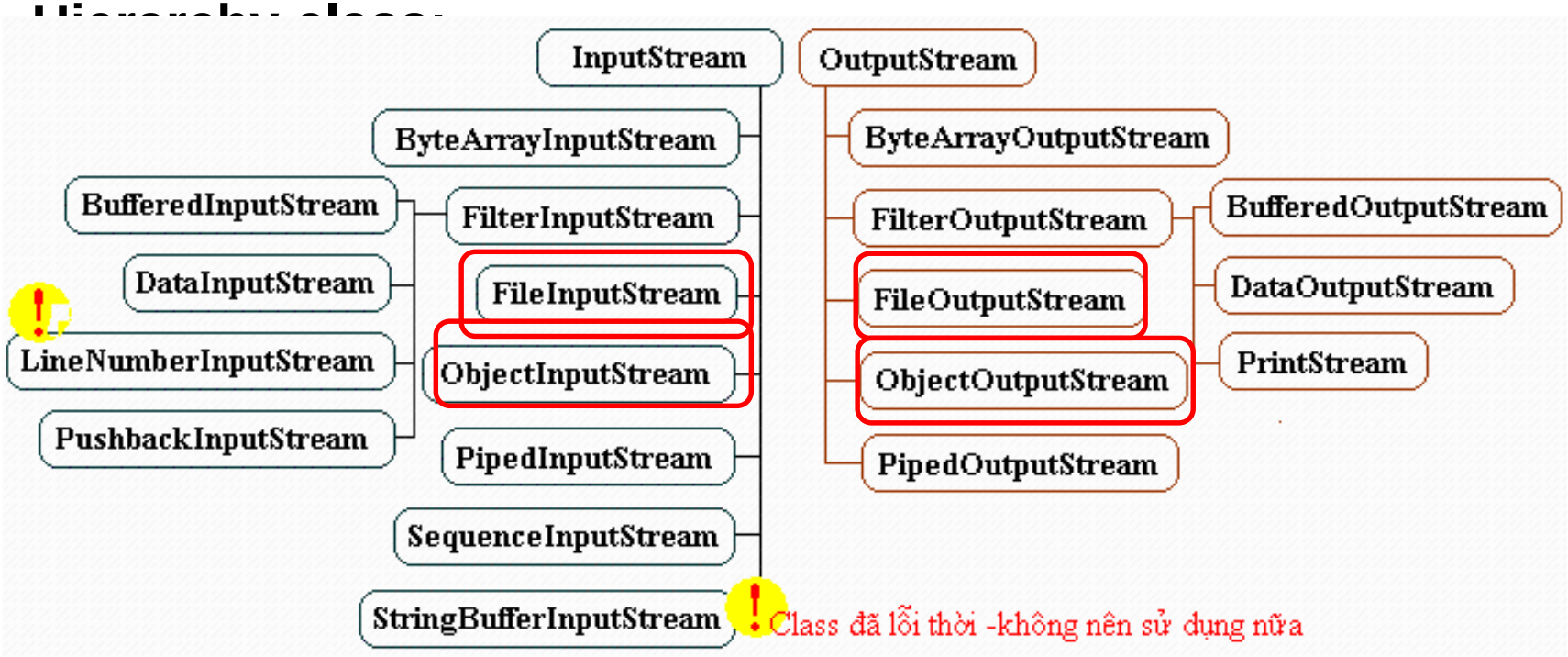
JAVA Stream

- A stream can be defined as a sequence of data. There are two kinds of Streams −
  - ✓ **InputStream** − The InputStream is used to read data from a source.
  - ✓ **OutputStream** − The OutputStream is used for writing data to a destination.



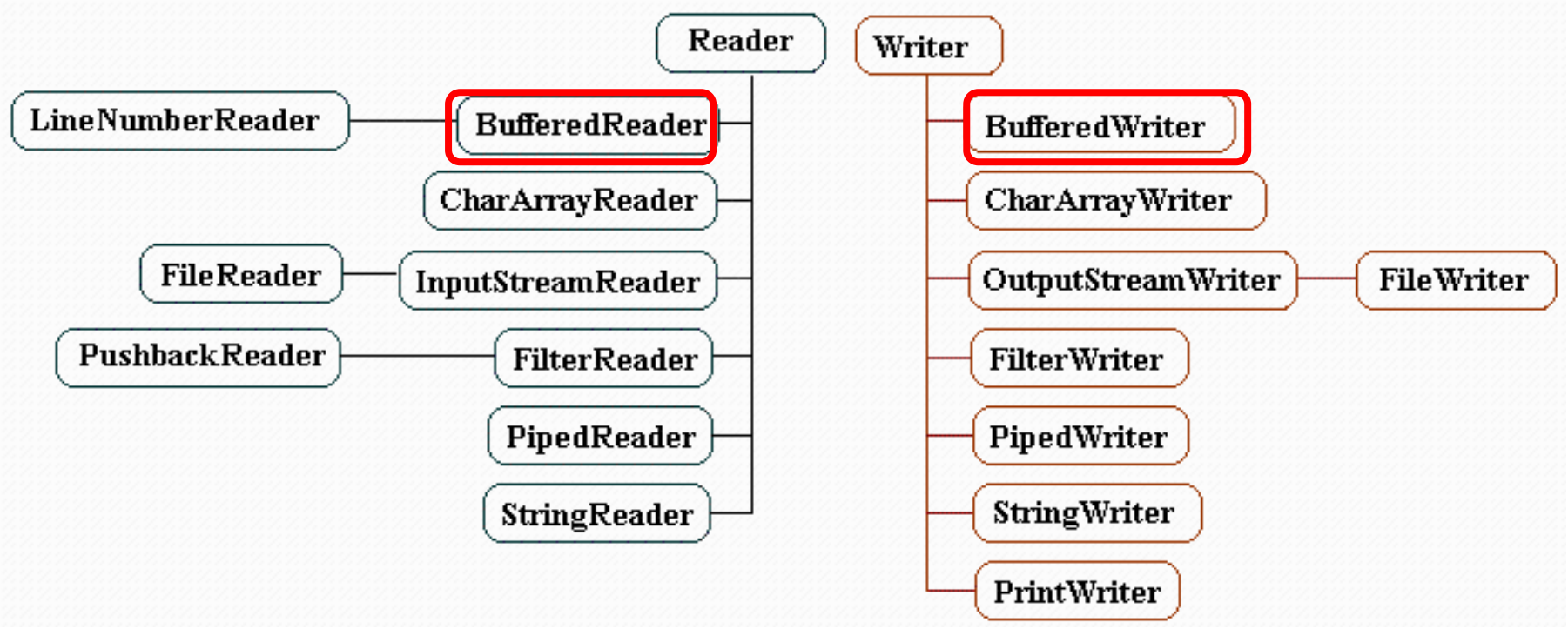
Binary streams

- Java byte streams are used to perform input and output of 8-bit bytes.
- Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**.
- **Hierarchy class:**



Text streams

- Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode.
- **Hierarchy class:**



FileInputStream class

- The stream is used for reading data from the files.
- **Class constructors:**

Sr.No.	Constructor & Description
1	**FileInputStream(File file)** This creates a FileInputStream by opening a connection to an actual file, the file named by the File object *file* in the file system.
2	**FileInputStream(FileDescriptor fdObj)** This creates a FileInputStream by using the file descriptor *fdObj*, which represents an existing connection to an actual file in the file system.
3	**FileInputStream(String name)** This creates a FileInputStream by opening a connection to an actual file, the file named by the path name *name* in the file system.

FileInputStream class

- Important methods:

Sr.No.	Method & Description
1	**void close()** This method closes this file input stream and releases any system resources associated with the stream.
2	int read() This method reads a byte of data from this input stream. **Returns**: the next byte of data, or -1 if the end of the file is reached.
3	int read(byte[] b) This method reads up to b.length bytes of data from this input stream into an array of bytes. **Parameters:** b - the buffer into which the data is read. **Returns**: the total number of bytes read into the buffer, or -1.
4	int read(byte[] b, int off, int len) This method reads up to len bytes of data from this input stream into an array of bytes. **Parameters:** b - the buffer into which the data is read. off - the start offset in the destination array b len - the maximum number of bytes read. **Returns:** the total number of bytes read into the buffer, or -1

FileInputStream class

- Important methods:

Sr.No.	Method & Description
1	**int available()**This method returns an estimate of the number of remaining bytes that can be read (or skipped over) from this input stream without blocking by the next invocation of a method for this input stream.

- Example:

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

    // attach the file to FileInputStream
    FileInputStream fin = new FileInputStream("data.txt");

    // illustrating available method
    System.out.println("Number of remaining bytes:" + fin.available());

    // illustrating skip method
    /*Original File content:
    * This is my first line
    * This is my second line*/
    fin.skip(5);
    System.out.println("FileContents :");
    // read characters from FileInputStream and write them
    int ch;
    while ((ch = fin.read()) != -1)
      System.out.print((char) ch);

    // close the file
    fin.close();
  }
}
```

```
Number of remaining bytes:46
FileContents :
is my first line
This is my second line
```

FileOutputStream class

- FileOutputStream class belongs to byte stream and stores the data in the form of individual bytes.
- **Class constructor:**

Sr.No.	Constructor & Description
1	**FileOutputStream(File file)** This creates a file output stream to write to the file represented by the specified *File* object.
2	**FileOutputStream(File file, boolean append)** This creates a file output stream to write to the file represented by the specified File object.
3	**FileOutputStream(FileDescriptor fdObj)** This creates an output file stream to write to the specified file descriptor, which represents an existing connection to an actual file in the file system.
4	**FileOutputStream(String name)** This creates an output file stream to write to the file with the specified name.
5	**FileOutputStream(String name, boolean append)** This creates an output file stream to write to the file with the specified name.

FileOutputStream class

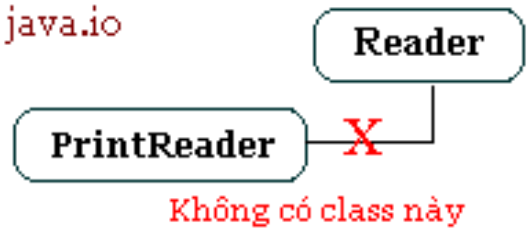
- **Important methods:**
  - ✓ **void close() :** Closes this file output stream and releases any system resources associated with this stream.
  - ✓ **protected void finalize() :** Cleans up the connection to the file, and ensures that the close method of this file output stream is called when there are no more references to this stream.
  - ✓ **void write(byte[] b) :** Writes b.length bytes from the specified byte array to this file output stream.
  - ✓ **void write(byte[] b, int off, int len) :** Writes len bytes from the specified byte array starting at offset off to this file output stream.
  - ✓ **void write(int b) :** Writes the specified byte to this file output stream.

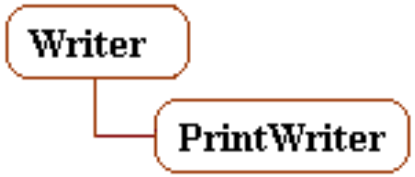
FileOutputStream class

- **Examples:**

```
public class WriteFile {
  public static void main(String[] args) throws IOException
  {
      //attach keyboard to DataInputStream
      DataInputStream dis=new DataInputStream(System.in);

      // attach file to FileOutputStream
      FileOutputStream fout=new FileOutputStream("file.txt");

      //attach FileOutputStream to BufferedOutputStream
      BufferedOutputStream bout=new BufferedOutputStream(fout,1024);
      System.out.println("Enter text (@ at the end):");
      char ch;

      //read characters from dis into ch. Then write them into bout.
      //repeat this as long as the read character is not @
      while((ch=(char)dis.read())!='@')
      {
          bout.write(ch);
      }
      //close the file
      bout.close();
  }
}
```

PrintWriter class





- **Class constructor:**

```
PrintWriter(Writer out)
 → Can append
PrintWriter(Writer out, boolean autoFlush)
PrintWriter(OutputStream out)
PrintWriter(OutputStream out, boolean autoFlush)
PrintWriter(String fileName)
 → Cannot append.
```

PrintWriter class

- To open a text file for output: connect a text file to a stream for writing

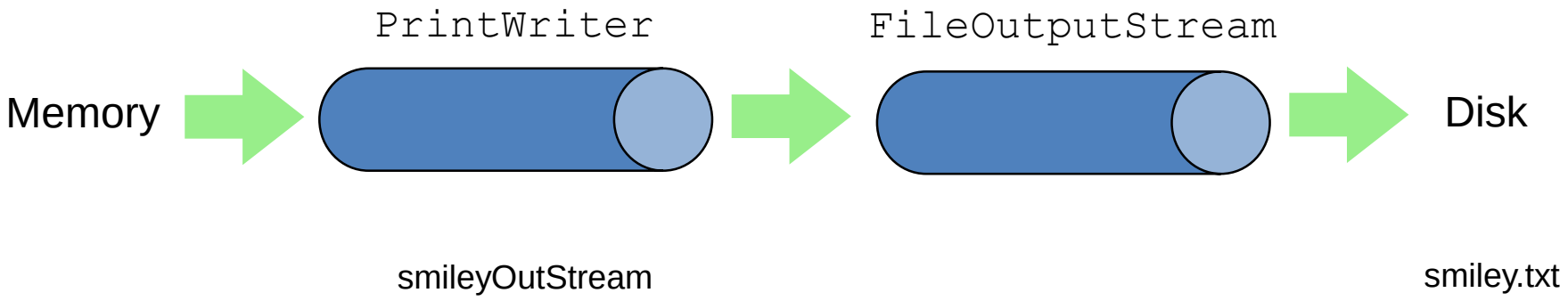
```
PrintWriter outputStream =
        new PrintWriter(new FileOutputStream("out.txt"));
```

- Similar to the long way:

```
FileOutputStream s = new
                    FileOutputStream("out.txt");
PrintWriter outputStream = new PrintWriter(s);
```

- **Goal**: create a `PrintWriter` object:
  - ✓ which uses `FileOutputStream` to open a text file.
- `FileOutputStream` "connects" `PrintWriter` to a text file.

Output File Streams




```
PrintWriter smileyOutStream = new PrintWriter( new FileOutputStream("smiley.txt") );
```

Java Tip: Appending to a Text Fi

- To add/append to a file instead of replacing it, use a different constructor for FileOutputStream:

```
outputStream =
new PrintWriter(new FileOutputStream("out.txt", true));
System.out.println("A for append or N for new file:");
char ans = Scanner.next().charAt(0);
boolean append = (ans == 'A' || ans == 'a');
outputStream = new PrintWriter(
new FileOutputStream("out.txt", append));
```

true if user enters 'A'

Closing a File

- An output file should be closed when you are done writing to it.
- An input file should be closed when you are done reading from it.
- Use the `close` method of the class `PrintWriter`, `BufferedReader`.

```
outputStream.close();
```

- If a program ends normally it will close any files that are open.

BufferReader class

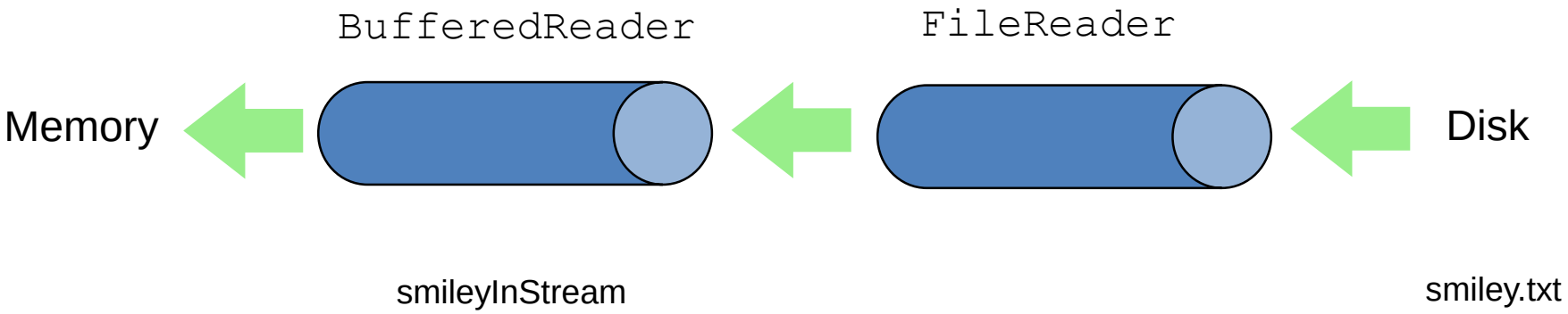
- To open a text file for input: connect a text file to a stream for reading
  - ✓ Goal: a `BufferedReader` object,
    - which uses `FileReader` to open a text file
  - ✓ `FileReader` "connects" `BufferedReader` to the text file
- For example:

```
BufferedReader smileyInStream = new BufferedReader
                    (new FileReader("smiley.txt"));
```

- Similarly, the long way:

```
FileReader s = new FileReader("smiley.txt");
BufferedReader smileyInStream=new BufferedReader(s);
```

Input File Streams




```
BufferedReader smileyInStream = new BufferedReader( new FileReader("smiley.txt") );
```

ObjectOutputStream class

- An **ObjectOutputStream** writes primitive data types and graphs of Java objects to an OutputStream.
  - ✓ Only objects that support the **java.io.Serializable** interface can be written to streams.
  - ✓ The Java **ObjectOutputStream** is often used together with a Java **ObjectInputStream**.
  - ✓ The ObjectOutputStream is used to write the Java objects, and the ObjectInputStream is used to read the objects again.
- **Class constructors :**
  - ✓ **protected ObjectOutputStream() :** Provide a way for subclasses that are completely reimplementing ObjectOutputStream to not have to allocate private data just used by this implementation of ObjectOutputStream.
  - ✓ **ObjectOutputStream(OutputStream out) :** Creates an ObjectOutputStream that writes to the specified OutputStream.

ObjectOutputStream class

- Important methods:
  - ✓ **void writeObject(Object obj) :** Write the specified object to the ObjectOutputStream.
- Examples:

```
public class ObjectOutputStreamDemo {
  public static void main(String[] args) throws IOException,
      ClassNotFoundException {
    FileOutputStream fout = new FileOutputStream("file.txt");
    ObjectOutputStream oot = new ObjectOutputStream(fout);
    String a = "Fresher Academy";
    String b = "Fresher";
    byte[] be = { 'A', 'B', 'C' };
    // illustrating writeInt(int i)
    oot.writeInt(1);
    // illustrating writeBoolean(boolean a)
    oot.writeBoolean(true);
    // illustrating writeObject(Object x)
    oot.writeObject(a);
    // illustrating writeBytes(String b)
    oot.writeBytes(b);
    // illustrating writeDouble(double d)
    oot.writeDouble(2.3);
    // illustrating writeUTF(String str)
    oot.writeUTF(a);
    // illustrating writeChars(String a)
    oot.writeChars(a);
    // illustrating write(byte[] buff)
    oot.write(be);
    // flushing the stream
    oot.flush();
    oot.close();
  }
}
```

ObjectInputStream class

- The **ObjectInputStream** class deserializes primitive data and objects previously written using an ObjectOutputStream. Following are the important points about BufferedInputStream:
  - ✓ It is used to recover those objects previously serialized. It ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine.
  - ✓ Classes are loaded as required using the standard mechanisms.

ObjectInputStream class

- **Examples:**

```
public class ObjectInputStreamDemo {
  public static void main(String[] args) throws IOException, ClassNotFoundException {
    byte[] be = { 'A', 'B', 'C' };
    byte c[] = new byte[4];

    FileInputStream fin = new FileInputStream("file.txt");
    ObjectInputStream oit = new ObjectInputStream(fin);
    System.out.println(oit.readInt());
    System.out.println(oit.readBoolean());
    System.out.println(oit.readObject());
    oit.read(c);
    for (int i = 0; i < 4; i++) {
      System.out.print((char) c[i]);
    }
    System.out.println();
    System.out.println(oit.readDouble());
    for (int i = 0; i < 13; i++) {
      System.out.print(oit.readChar());
    }
    System.out.println();
    System.out.println(oit.readShort());
    oit.readFully(be);

    for (int i = 0; i < 3; i++) {
      System.out.print((char) be[i]);
    }
    oit.close();
  }
}
```

Section 2

EXCEPTION HANDLING

Introduction exception

- ✓ During the execution of a program, the computer will face the some of situations:
  - syntax error
  - logic algorithm error
  - runtime error
- ✓ An exception is an abnormal condition that arises[xuất hiện] in a code sequence at run time.
  - an exception is a run-time error
- ✓ Java's exception handling avoids the problems in the run-time.

Introduction exception (cont.)

- ✓ A program that requests a number from the user:
- ✓ Ex 1:

```
public class Exercise {
    public static void main(String[] args) {
        double side;
        Scanner scnr = new Scanner(System.in);
        System.out.println("Square Processing");
        System.out.print("Enter Side: ");
        side = scnr.nextDouble();
        System.out.println("\nSquare Characteristics");
        System.out.printf("Side: %.2f\n", side);
        System.out.printf("Perimeter: %.2f\n", side * 4);
    }
}
```

```
Square Processing
Enter Side: dS
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextDouble(Unknown Source)
        at clc.btjb.unit04.les04.Exercise.main(Exercise.java:11)
```

Introduction exception (cont.)

- ✓ Runtime errors can be divided into low-level errors that involve violating constraints, such as:
  - dereference of a null pointer
  - out-of-bounds array access
  - divide by zero
  - attempt to open a non-existent file for reading
  - bad cast (e.g., casting an Object that is actually a Boolean to Integer)

Introduction exception (cont.)

- ✓ and higher-level, logical errors, such as violations of a function's precondition:
  - call to Stack's "pop" method for an empty stack
  - call to "factorial" function with a negative number
  - call to List's nextElement method when hasMoreElements is false

Introduction exception (cont.)

- **Ex 2:**

```
public class Lowlevel_Exception {
    public static void main(String[] args) {
        int x = 0, z = 5;
        int j = 3;
        int y = z / x;  // Exception: Divide By Zero

        int[] ar = new int[j];// Exception: Index Out Of Range
        ar[j] = 5;
        // suppose that a.txt does not exist
    BufferedReader input = new BufferedReader(
                                new FileReader("a.txt"));
    // Exception: File Not Found
    }
}
```

Exception handling

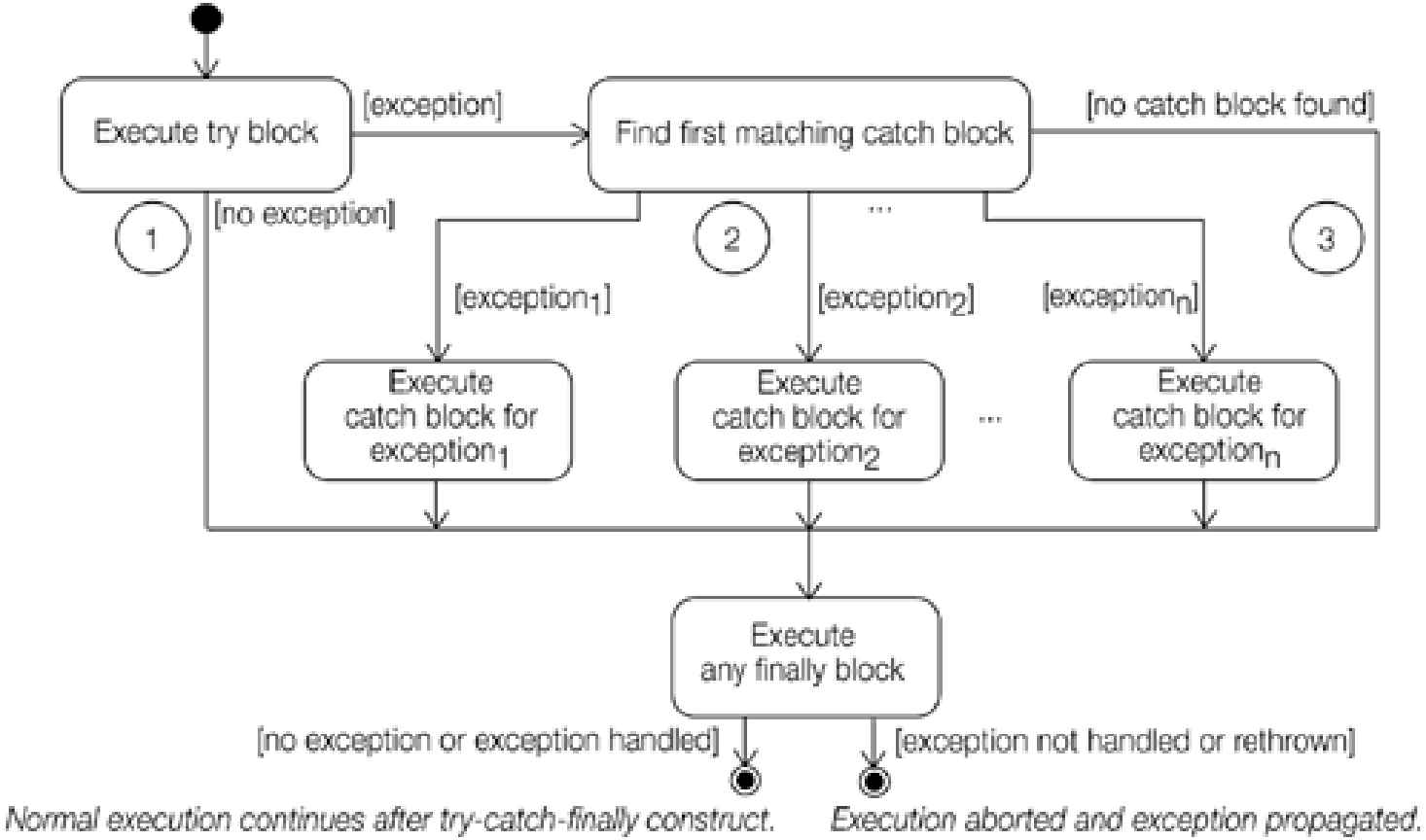
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
  - Program statements that you want to monitor for exceptions are contained within a **try** block.
  - Your code can catch this exception (using **catch**) and handle it in some rational manner.
  - To manually throw an exception, use the keyword **throw**. Any exception that is thrown out of a method must be specified as such by a **throws** clause. Any code that absolutely must be executed before a method returns is put in a **finally** block.

Exception handling (cont.)

- ✓ **This is the general form of an exception-handling block:**

```
try {
   // block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
   // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
   // exception handler for ExceptionType2
}
// ...
[finally {
   // block of code to be executed before try block ends
}]
```

Exception handling (cont.)



Exception handling (cont.)

- ✓ **Solution ex1:**

```
try{
    side = scnr.nextDouble();
    System.out.println("\nSquare Characteristics");
    System.out.printf("Side: %.2f\n", side);
    System.out.printf("Perimeter: %.2f\n", side * 4);
}
catch(InputMismatchException ex)
{
  System.out.println(ex.getMessage());
  // method get message
}
```

Exception handling (cont.)

- ✓ **Solution ex2:**

```
public static void main(String[] args) {
    int x = 0, z = 5;
    int j = 3;
    int[] ar = new int[j];
    try {
        int y = z / x;
        ar[j] = 5;

        BufferedReader input = new BufferedReader(new
                                FileReader("a.txt"));
    } catch (Exception ex) {
       ex.getStackTrace();
      // Exception Method Call Stack Trace
    }
}
```

Exception handling (cont.)

- ✓ **Solution ex2:**

```
} catch (ArithmeticException ex1) {
  System.out.println(ex1.getStackTrace());
}
catch(ArrayIndexOutOfBoundsException ex2)
{
  System.out.println(ex2.getStackTrace());
}
catch(FileNotFoundException ex3)
{
  System.out.println(ex3.getStackTrace());
}
```

Exception handling (cont.)

✓ **Finally block**

```
Connection conn= null;
try {
  conn= get the db conn;
  //do some DML/DDL
} catch(SQLException ex)
{
} finally
{
  conn.close();
}
```

Using Throws and Throw Statements

- **Ex 3:** Imagine you have been assigned a task of finding a specific book, and then reading and explaining its contents to a class of students. The required sequence may look like:
  - ✓ Get the specified book
  - ✓ Read aloud its contents
  - ✓ Explain the contents to a class of students.
- **Proplem:** But what happens if you **can't find** the specified book? You can't proceed with the rest of the action without it so you **need to report back** to the person who assigned the task to you. This unexpected event (missing book) prevents you from completing your task. By reporting it back, you want the originator of this request to take corrective or alternate steps.

Using Throws and Throw Statement

- **Solution:**

Keyword throws

```
void teachClass() throws BookNotFoundException {

    boolean bookFound = locateBook();

    if (!bookFound)
        throw new BookNotFoundException();
    else {
        readBook();
        explainContents();
    }

}
```

Keyword throw

Using Throws and Throw Statements (cont.)

- **Creating a method that throws a checked exception**

```
public class DemoThrowsException {
    public void readFile(String file) throws          //#1
                FileNotFoundException {
     boolean found = findFile(file);
     if (!found)
     throw new FileNotFoundException("Missing file"); //#2
     else {
                //code to read file
            }
        }
    boolean findFile(String file) {
    // code to return true if file can be located
    }
}
```

Using Throws and Throw Statements (cont.)

- **In which:**
  - **#1:** The throws statement indicates that this method can throw `FileNotFoundException`
  - **#2:** If file can't be found, code creates and throws an object of FileNotFoundException by using the **`throw`** statement
  - A method can include names of multiple, **comma** separated names in its throws statement
- **Using a method that throws a checked exception**

  When you use a method that throws a checked exception
  - Enclose the code within a try block and catch the thrown exception.
  - Declare the exception to be rethrown in the method's signature.
  - Implement both the above together.

Using Throws and Throw Statements (cont.)

- **Enclose the code within a try block and catch the thrown exception:**

```
void useReadFile(String name) {
    try {
      readFile(name);
    } catch (FileNotFoundException e) {
      // code
    }
}
```

- **Declare the exception to be rethrown in the method's signature:**

```
void useReadFile(String name) throws
                FileNotFoundException{
          readFile(name);
}
```

Using Throws and Throw Statements (cont.)

- **Implement both the above together:**

```
void useReadFile(String name) throws
                FileNotFoundException {
    try {
      readFile(name);
    } catch (FileNotFoundException e) {
      // code
    }
}
```

Checked vs Unchecked exceptions

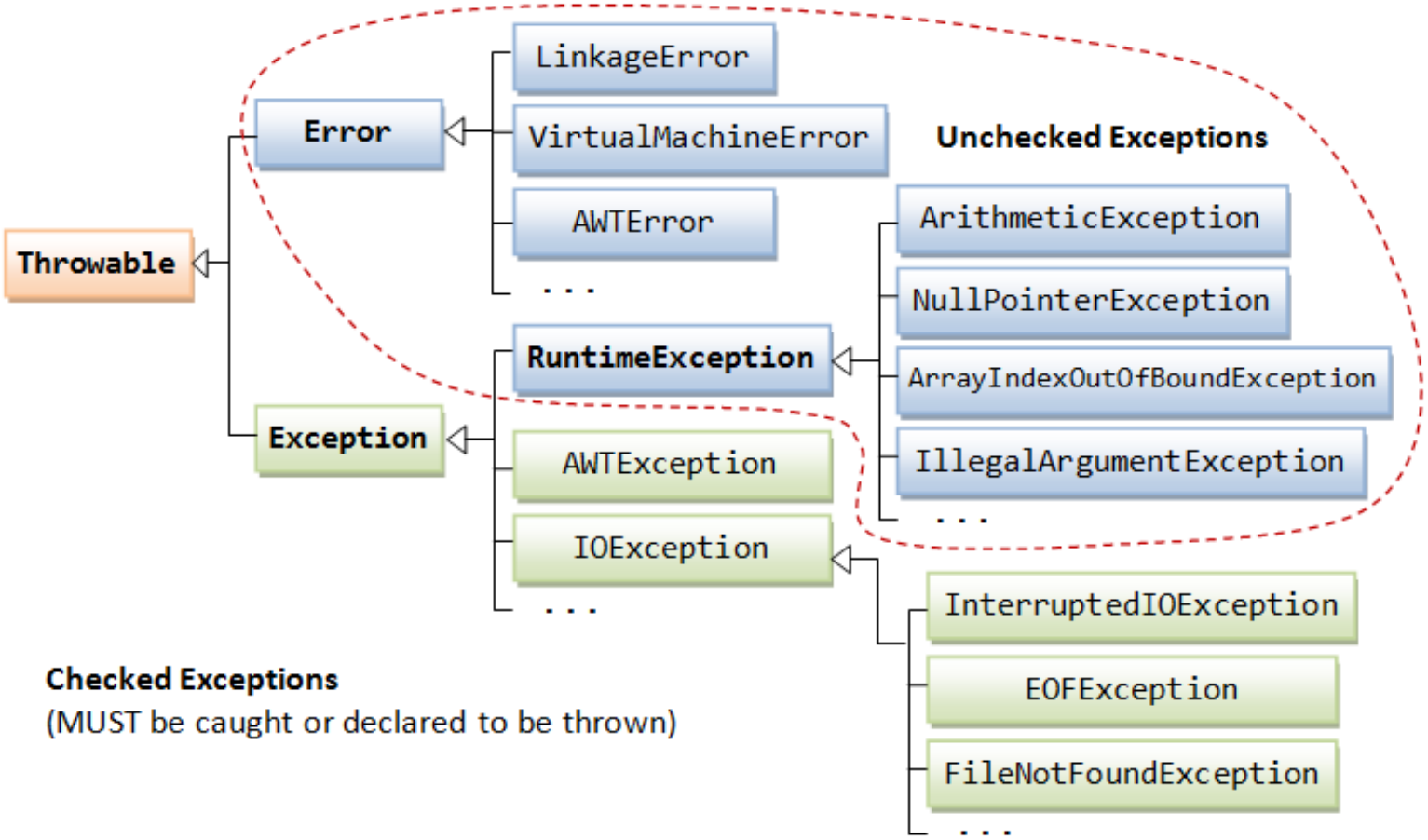
- **Checked exceptions**
  - All exceptions other than Runtime Exceptions are known as Checked exceptions as the **compiler checks them during compilation** to see whether the programmer has handled them or not.
  - If these exceptions are not handled/declared in the program**, it will give compilation error**.
  - **Examples of Checked Exceptions:**
    - `ClassNotFoundException`
    - `IllegalAccessException`
    - `NoSuchFieldException`
    - `EOFException, etc.`
- **See ex1**

Checked vs Unchecked exceptions (cont.)

- **Unchecked Exceptions**
  - Runtime Exceptions are also known as Unchecked Exceptions as the **compiler do not check whether** the programmer has handled them or not but it's the duty of the programmer to handle these exceptions and provide a safe exit.
  - If these exceptions are not handled/declared in the program**, it will not give compilation error**.
  - **Examples of UnChecked Exceptions:**
    - `ArithmeticException`
    - `ArrayIndexOutOfBoundsException`
    - `NullPointerException`
    - `NegativeArraySizeException, etc.`
- **See ex1**

Exception classes

- The figure below shows the hierarchy of the Exception classes.
  - The base class for all Exception objects is: `java.lang.Throwable`, `java.lang.Exception` and `java.lang.Error`.



Summary

- **Java IO Basic**
- **Exception Handling**

Thank you