

CHAPTER 4

Exception Handling – Files and IO

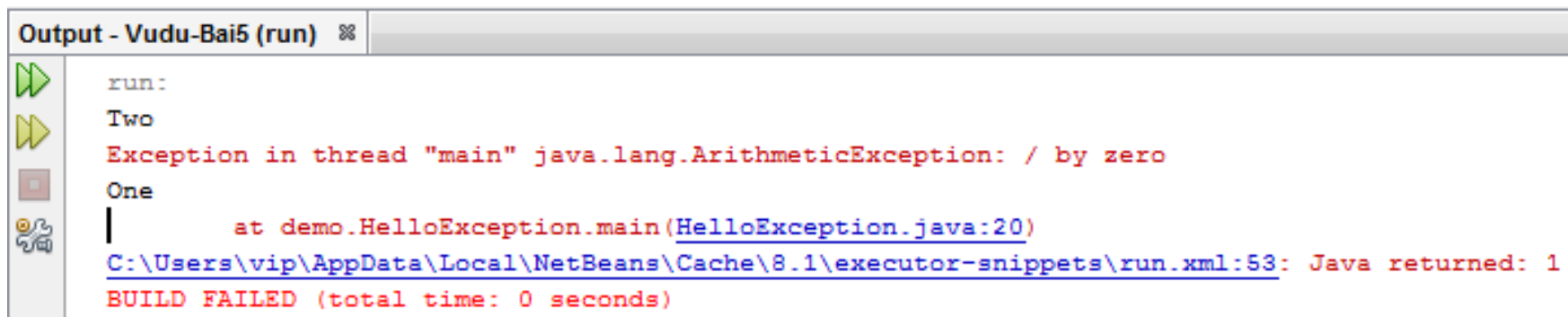
Objectives

- Example for exceptions
- What is an Exception?
- Exception class
- Stack Traces
- Kinds of Exception
- Creating Your Own Exception Classes
- Java input and output (I/O) overview (Distinguishing Text, UTF, and Unicode)
- How to access text files.
- How to access binary files?
- How to read/write objects from/to files
- Case study

Exception Handling

Example

```
public class HelloException {  
    public static void main(String[] args) {  
        System.out.println("Two");  
        // The division is ok  
        int value = 10 / 2;  
        System.out.println("One");  
        // divide by zero  
        // error encountered in here  
        value = 10 / 0;  
        //this line is not executed  
        System.out.println("Let's go!");  
    }  
}
```



```
Output - Vudu-Bai5 (run) ✖  
run:  
Two  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
One  
|       at demo.HelloException.main(HelloException.java:20)  
C:\Users\vip\AppData\Local\NetBeans\Cache\8.1\executor-snippets\run.xml:53: Java returned: 1  
BUILD FAILED (total time: 0 seconds)
```

Fix it

```
public static void main(String[] args) {  
    System.out.println("Two");  
    // The division is ok  
    int value = 10 / 2;  
    System.out.println("One");  
    // divide by zero and error encountered in here  
    try{  
        value = 10 / 0;  
    }catch(ArithmeticException e) {  
        System.out.println(e);  
    }  
    //this line is executed  
    System.out.println("Let's go!");  
}
```

The computer takes exception

- Exceptions are errors in the logic of a program (run-time errors).
- Examples:
- Exception in thread “main” java.io.FileNotFoundException: student.txt (The system cannot find the file specified.)
- Exception in thread “main” java.lang.NullPointerException: at FileProcessor.main(FileProcessor.java:9)
- Question: do all run-time errors cause Exceptions?

Exceptions

- When a program is executing something occurs that is not quite normal from the point of view of the goal at hand.
- For example:
 - a user might type an invalid filename;
 - a file might contain corrupted data;
 - a network link could fail;
 - ...
- Circumstances of this type are called exception conditions in Java and are represented using objects (All exceptions descend from the `java.lang.Throwable`).

The Exception Class

- As with anything in Java, Exception is a class

Method	What it does
<code>void printStackTrace()</code>	Prints the sequence of method calls leading up to the statement that caused the Exception.
<code>String getLocalizedMessage()</code>	Returns a “detail” message.
<code>String toString()</code>	Returns the Exception class name and detail message.

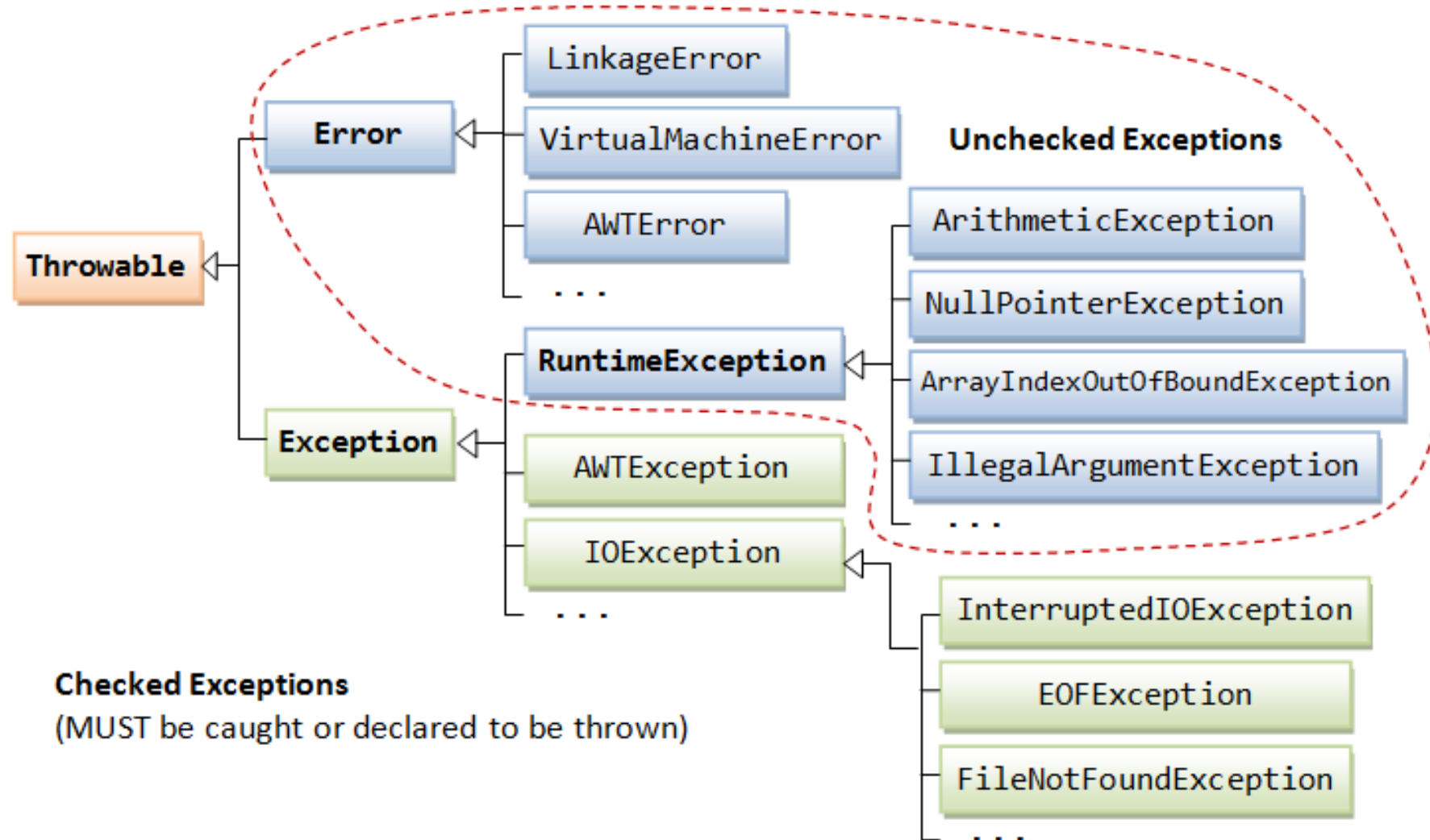
- The methods in the Exception class can be useful for debugging, as we will see.

Catching Exceptions

```
try {  
    // Exception-throwing code  
}  
catch (Exception_type name) {  
    // Exception-handling code  
}
```

- If no exception occurs during the try block:
 - jump to statements after all the catch blocks.
- If an exception occurs in the try block:
 - jump to the first handler for that type of exception.
 - After the catch finishes, jump to the statement after all the catch blocks.

Exception Classes



Example

```
public class Exception1 {  
    public static void main(String[] args) {  
        String sNum = "CTB";  
        String sDate = "10/03/2016";  
        int num = Integer.parseInt(sNum);  
        SimpleDateFormat f = new SimpleDateFormat("dd/MM/yyyy");  
        Date d = f.parse(sDate);  
        String s = "Subject - OOP";  
        System.out.println(s.substring(50));  
    }  
}
```

Stack Traces

- How do you know what went wrong?
- All exceptions have methods that return information about the cause of the Exception:

Method	Description
<code>getLocalizedMessage()</code>	Returns a String containing a description of the error
<code>getStackTrace()</code>	Returns an array of <code>StackTraceElement</code> objects, each of which contains info about where the error occurred
<code>printStackTrace()</code>	Displays the Stack Trace on the console.

Displaying the stack trace info

```
import java.util.*; // For Scanner class
import java.io.*;
public class FileProcessor{
    public static void main(String [] args) {
        try {
            File inputFile = new File("student.txt");
            Scanner input = new Scanner(inputFile);
            while(input.hasNextLine()) {
                System.out.println("> " + input.nextLine());
            }
        }
        catch(FileNotFoundException exception) {
            System.out.println("Could not find the file 'student.txt'.");
            System.out.println(exception.getLocalizedMessage());
            exception.printStackTrace();
        }
    }
}
```

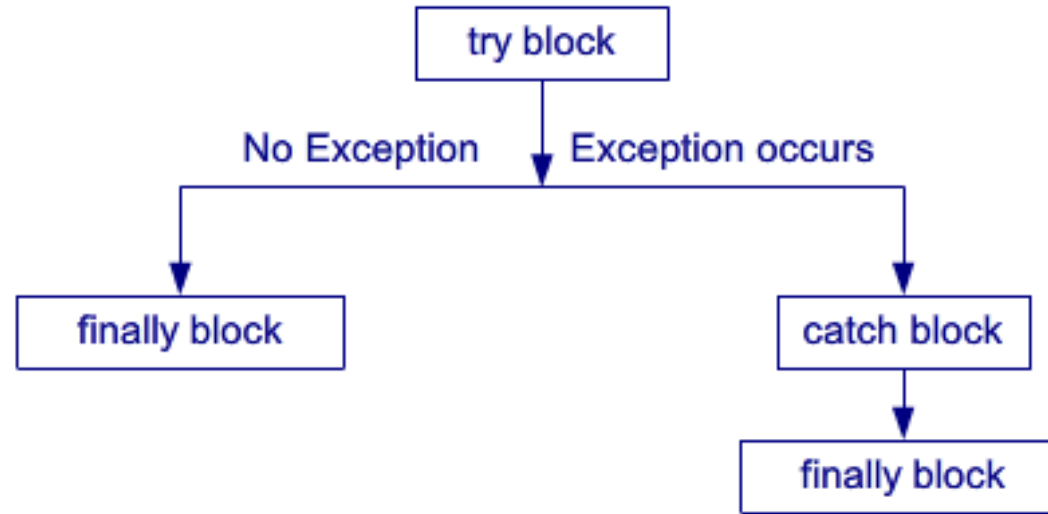
Multiple catch blocks

- try block may contain code that throws different exception types. This can even happen if the block contains only a single line of code, because a method is allowed to throw different types to indicate different kinds of trouble.

```
try {  
    // do something ...  
} catch (Exception1 e) {  
    //Exception handler for Exception1  
} catch (Exception2 e) {  
    //Exception handler for Exception2  
} finally {  
    // this finally always executed  
    // to do something here  
}
```

The finally Block

- The last catch block associated with a try block may be followed by a finally block.



- The finally block's code is guaranteed to execute in nearly all circumstances excepts:
 - The death of the current thread
 - Execution of `System.exit()`
 - Turning off the computer

Declaring Exceptions

- There is a way to call exception-throwing methods without enclosing the calls in try blocks. A method declaration may end with the throws keyword, followed by an exception type, or by multiple exception types followed by commas.

```
private int throwsTwo() throws IOException, AWTException  
{...}
```

Of course, methods that call throwsTwo() must either enclose those calls in try blocks or declare that they, too, throw the exception types.

Two Kinds of Exception

- *Checked exception*
 - Must be handled by either the try-catch mechanism or the throws-declaration mechanism.
- Runtime exception (*Unchecked exception*)
 - The right time to deal with runtime exceptions is when you're designing, developing, and debugging your code. Since runtime exceptions should never be thrown in finished code.

There is some difference there for checked and unchecked Exception

Suppose **checked exception** means it shows the **compile time exception**

ex: NoSuchMethod exception, NoSuch Field Exception , ClassNot Found Exception

where as in **Unchecked Exception** shows **Runtime** Only

ex: Nullpointer Exception, Number Format Exception

Checked exceptions should always be caught

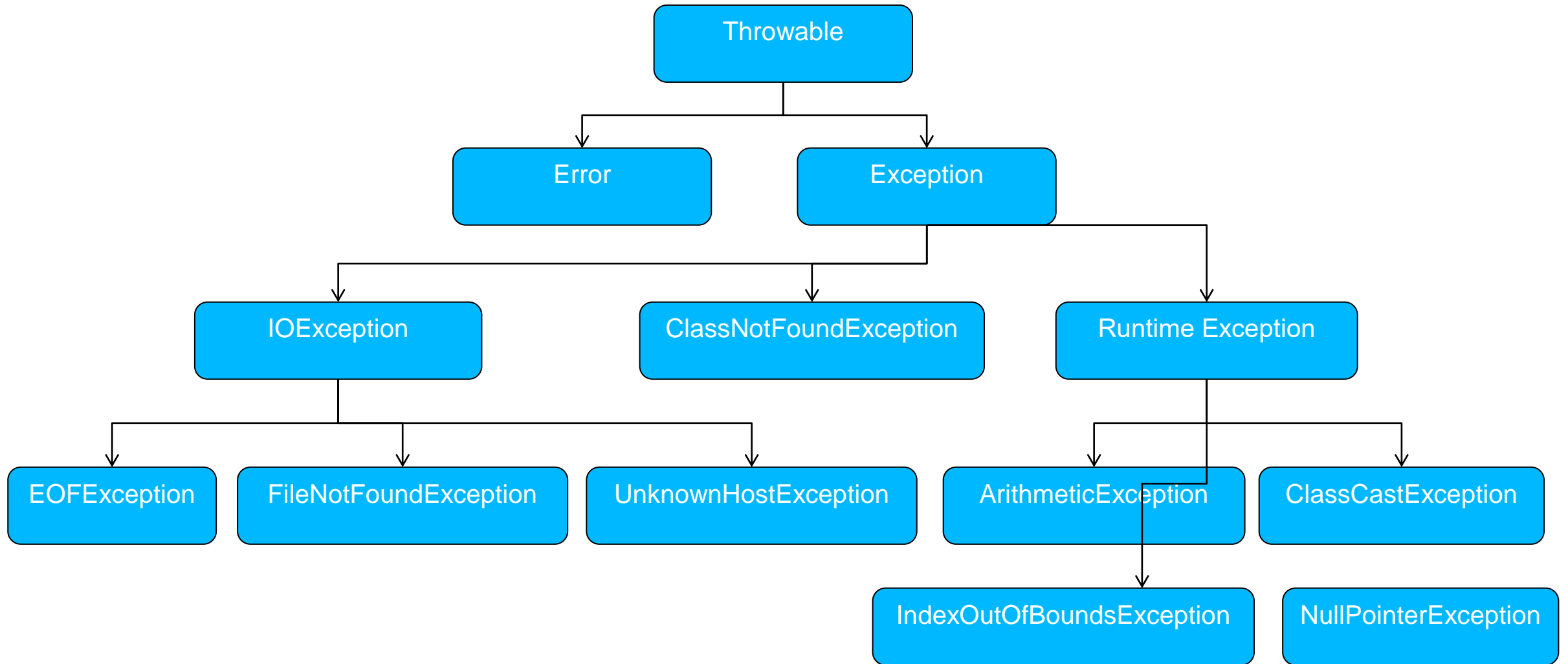
Runtime exceptions don't have to be caught.

Errors should never be caught.

The stack trace is recorded when the exception is constructed.

It is never appropriate for application programmers to construct and throw errors.

The Exception Inheritance Hierarchy



Some runtime (unchecked) exceptions

- The types of exceptions that need not be included in a methods throws list are called Unchecked Exceptions. They can be foreseen and prevented by a programmer and, generally speaking, will never occur in a high-quality program:
- `ArithmeticException`
- `ArrayIndexOutOfBoundsException`
- `ClassCastException`
- `IndexOutOfBoundsException`
- `IllegalStateException`
- `NullPointerException`
- `SecurityException`

Some checked exceptions

- The types of exceptions that must be included in a methods throws list if that method can generate one of these exceptions and does not handle it itself are called Checked Exceptions. Checked exceptions can happen at any time, cannot be prevented and therefore the language enforces to deal with them.
- ClassNotFoundException
- CloneNotSupportedException
- IllegalAccessException
- InstantiationException
- InterruptedException
- NoSuchFieldException
- NoSuchMethodException

```
public class Exception4 {  
    public static void main(String[]  
args){  
        File file = new File("example.txt");  
        try{  
            file.createNewFile();  
        } catch (IOException ex){  
            System.out.println(ex);  
        }  
    }  
}
```

throw and throws

- The throw keyword in Java is used to explicitly throw an exception from a method or any block of code. We can throw either checked or unchecked exception.
- throws is a keyword in Java which is used in the signature of method to indicate that this method might throw one of the listed type exceptions. The caller to these methods has to handle the exception using a try-catch block.

```

class ThrowExcepl {
    static void fun() {
        try{
            throw new
NullPointerException("demo");
        }
        catch(NullPointerException e) {
            System.out.println("Caught
inside fun().");
            throw e; // rethrowing the
exception
        }
    }
    public static void main(String
args[]) {
        try{
            fun();
        }
        catch(NullPointerException e) {
            System.out.println("Caught in
main.");
        }
    }
}

```

```

class ThrowsExecp2 {
    static void fun() throws
IllegalAccessException
    {
        System.out.println("Inside
fun(). ");
        throw new
IllegalAccessException("demo");
    }
    public static void main(String
args[]) {
        try{
            fun();
        }
        catch(IllegalAccessException e) {
            System.out.println("caught in
main.");
        }
    }
}

```

Creating Your Own Exception Classes

- Decide whether you want a checked or a runtime exception.
 - Checked exceptions should extend `java.lang.Exception` or one of its subclasses.
 - Runtime exceptions should extend `java.lang.RuntimeException` or one of its subclasses
- The argument list of these constructors should include
 - A message
 - A cause
 - A message and a cause

```
class MyException extends Exception {  
    MyException() {...}   
    MyException(String s) {  
        super(s);  
    }  
    ...  
}  
  
public class ExceptionDemo {  
    public static void executeHasException() throws MyException{  
        throw new MyException();  
    }  
}
```

Example (validate)

```
public class ValidateException extends Exception {
    public ValidateException(String message) {
        super(message);
    }
}

private void validateCode(String id)
    throws ValidateException{
    if(!id.matches("^ [Bb]{1} \\d{2} [A-Za-z]{4} \\d{3}$"))
        throw new ValidateException("Code \""+id+"\"invalid");
    }

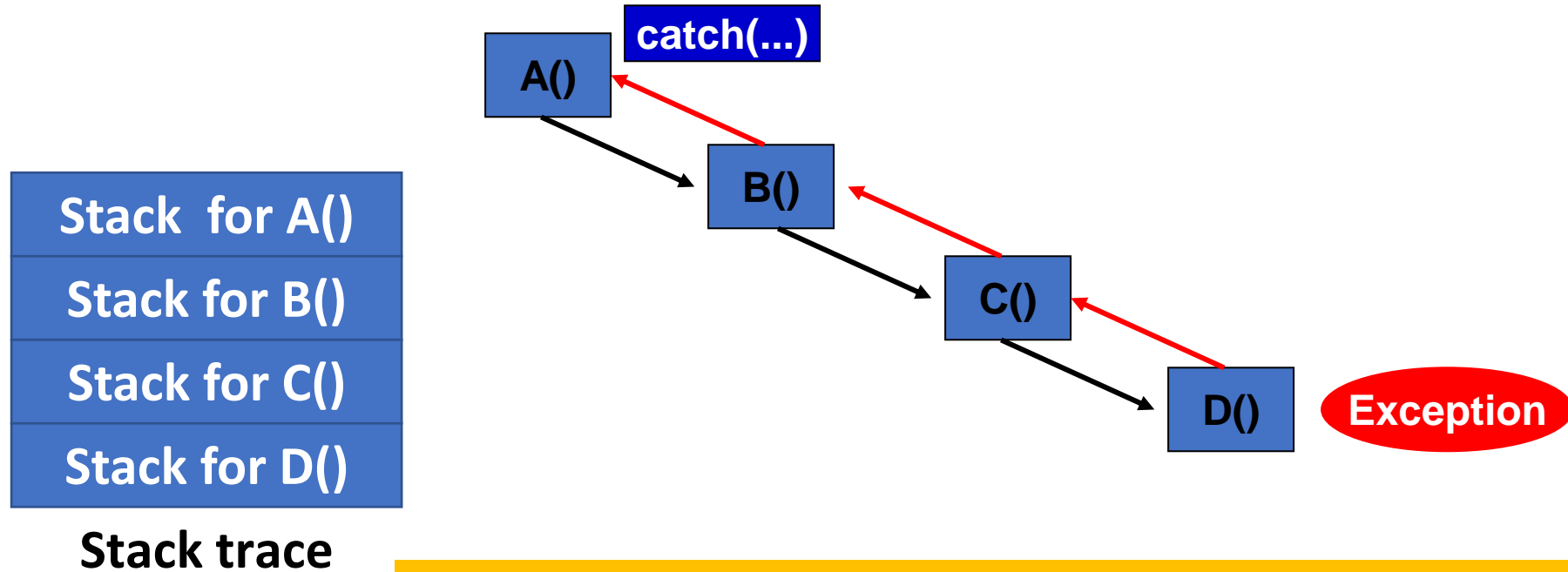
private void validateDob(String dob)
    throws ValidateException{
    if(!dob.matches("\\d{2}-\\d{2}-\\d{4}"))
        throw new ValidateException("DOB \""+dob+"\"invalid");
    }
```


Exceptions and Overriding

- When you extend a class and override a method, the Java compiler insists that all exception classes thrown by the new method must be the same as, or subclasses of, the exception classes thrown by the original method. In other words, an overridden method in a sub class must not throw Exceptions not thrown in the base class. Thus if the overriding method does not throw exceptions, the program will compile without complain.

```
class Disk {  
    void readFile() throws EOFException{}  
}  
class FloppyDisk extends Disk {  
    // ERROR!  
    void readFile() throws IOException {}  
}  
class DiskFix {  
    void readFile() throws IOException {}  
}  
class FloppyDisk extends Disk {  
    void readFile() throws EOFException {} //OK  
}
```

Exception Propagations



When an exception occurs at a method, program stack is containing running methods (method A calls method B,....). So, we can trace statements related to this exception.

Files and IO

Memory vs. Disk

- Disks have greater capacity (more GB) and offer permanent storage;
- Memory is much faster.

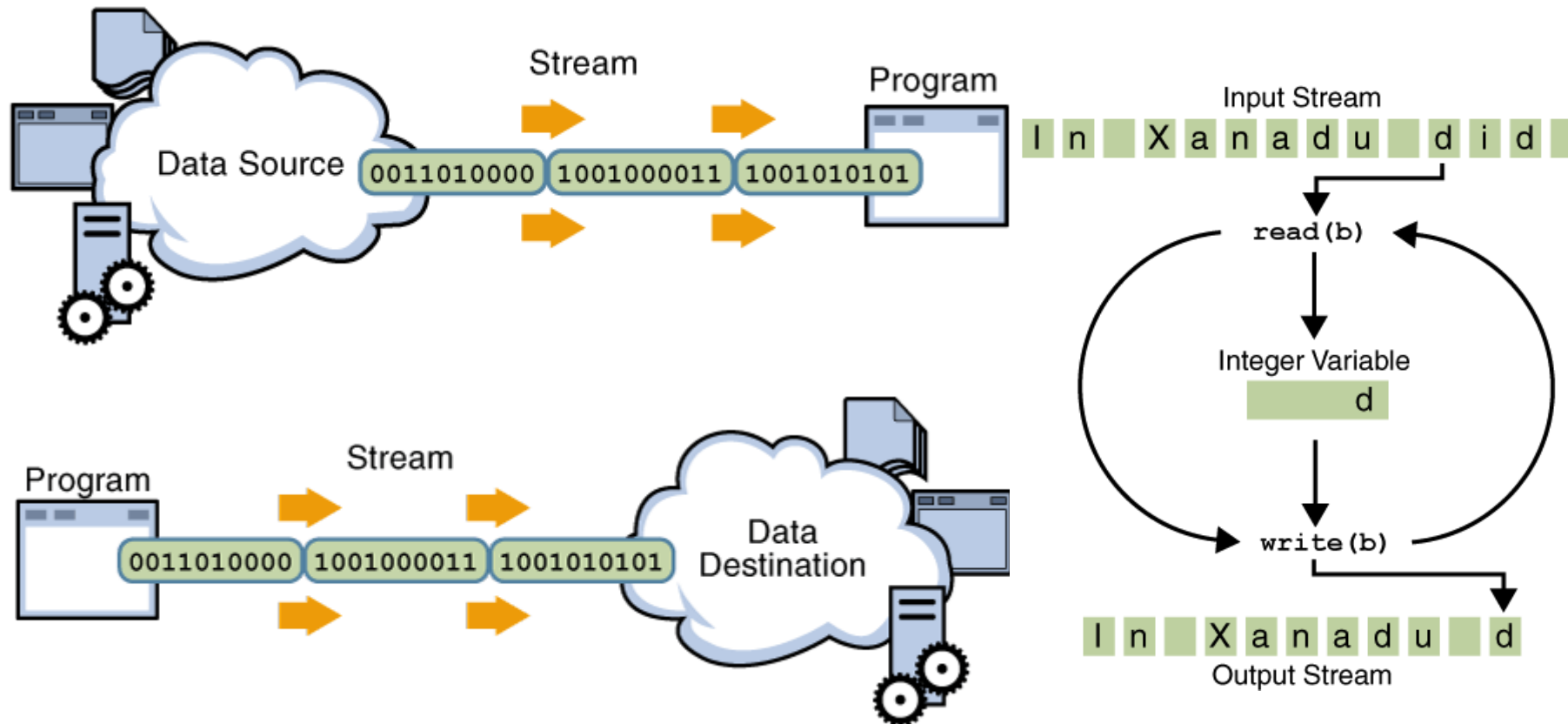
Typical Properties	Memory	Disk
Capacity:	1-4 Gb	>100 Gb
When power goes off:	Data is lost	Data is safe
When program ends:	Data is lost	Data is safe
Sequential access speed:	1.2 GB / second	50 MB / second
Random access speed:	1.2 GB / second	~ 66.7 seeks / second

Files and Variables

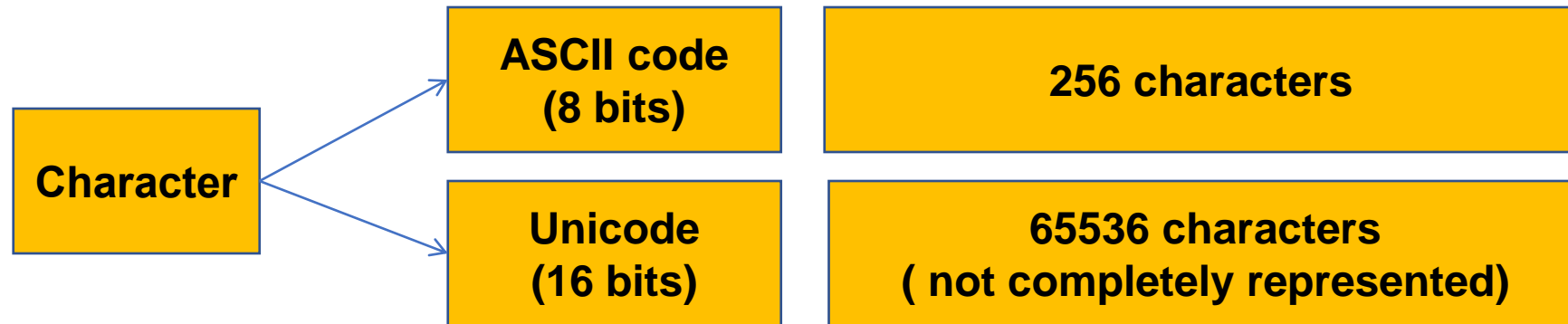
- Recall variables:
 - They have types, names, and a location in memory
 - You can put data inside them, and use the data later
- Files are similar abstractions, but for the disk:
- They have names and a location on the disk
 - You can put (lots of) data inside them
 - You can later retrieve that data and use it
- Read
 - Move data from a file on the disk into a variable (or variables) in memory
- Write
 - Move data from a variable (or variables) in memory to a file on the disk
- Two tricky details to these operations –
 - Data types
 - Checked Exceptions

What are streams?

- A stream is an object managing a data source in which operations such as read data in the stream to a variable, write values of a variable to the stream associated with type conversions are performed automatically. These operations treat data as a chain of units (byte/character/data object) and data are processed in unit-by-unit manner.



Text, UTF, and Unicode



Unicode character: a character is coded using 16/32 bits

UTF: Universal Character Set – UCS- Transformation Format

UTF: *Unicode transformation format* , a Standard for compressing strings of Unicode text .

UTF-8: A standard for compressing Unicode text to 8-bit code units.

Refer to: <http://www.unicode.org/versions/Unicode7.0.0/>

Java :

- Uses UTF to read/write Unicode
- Helps converting Unicode to external 8-bit encodings and vice versa.

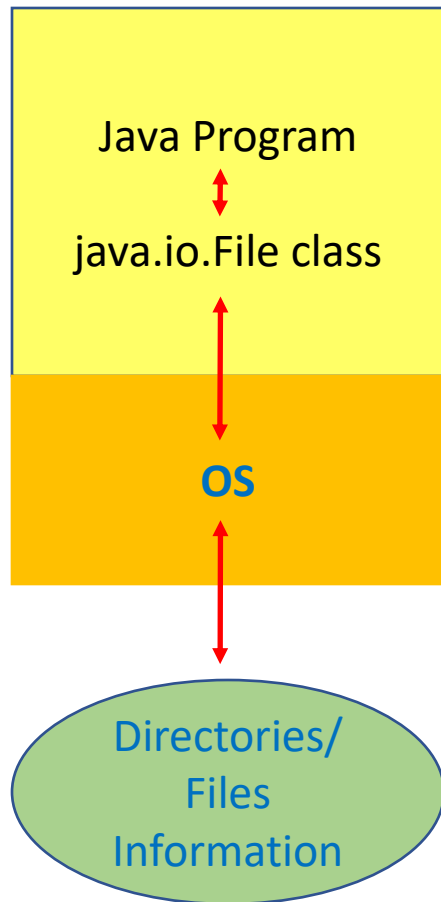
Introduction to the java.io Package

- Java treats all data sources (file, directory, IO devices,...) as streams
- The java.io package contains Java APIs for accessing to/from a stream.
- A stream can be a binary stream.
- Binary low-level stream: data unit is a physical byte.
- Binary high-level stream: data unit is primitive data type value or a string.
- Object stream: data unit is an object.
- A stream can be a character stream in which a data unit is an Unicode character.

Accessing directories and files (1)

The java.io.File Class

Class represents a file or a directory managed by operating system.



Constructor Summary

File(**File** parent, **String** child)

Creates a new File instance from a parent abstract pathname and a child pathname string.

File(**String** pathname)

Creates a new File instance by converting the given pathname string into an abstract pathname.

File(**String** parent, **String** child)

Creates a new File instance from a parent pathname string and a child pathname string.

File(**URI** uri)

Creates a new File instance by converting the given file: URI into an abstract pathname.

Accessing directories and files (2)

The java.io.File Class...

Common Methods:

boolean canExecute(), canRead(), canWrite()
boolean exists(), isDirectory(), isFile()
String getAbsolutePath(), getCanonicalPath(),
 getName(), getParent()
String[] list()
boolean delete(), createNewFile(), mkdir(),
 rename(File newName)
long length()

This class helps
accessing
file/directory
information only. It
does not have any
method to access
data in a file.

Method Invoked	Returns on Microsoft Windows
getAbsolutePath()	c:\java\examples\examples\student.txt
getCanonicalPath()	c:\java\examples\student.txt

Access Text Files

- Reading data from text files (Scanner)
- Character Streams:
- Two ultimate abstract classes of character streams are Reader and Writer.
- Reader: input character stream will read data from data source (device) to variables (UTF characters).
- Writer: stream will write UTF characters to data source (device).

Reading data from text files (Scanner)

- Creating a Scanner for a file, general syntax:

```
Scanner <name> = new Scanner(new File("<file name>"));
```

```
Scanner input = new Scanner(new File("numbers.txt"));
```

- Instead of getting data from the keyboard via `System.in`, this Scanner object gets data from the file `numbers.txt` in the current folder (directory).
- Each call to `next`, `nextInt`, `nextDouble`, etc. advances the cursor to the end of the current token, skipping over any whitespace. Each call consumes the input.

```
input.nextDouble();
```

```
308.2\n    14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n    ^
```

```
input.nextDouble();
```

```
308.2\n    14.9 7.4  2.8\n\n\n3.9 4.7 -15.4\n2.8\n        ^
```

Exercise 1

- Consider an input file named input.txt:

308.2

14.9 7.4 2.8

3.9 4.7 -15.4

2.8

- Write a program that reads the first 5 values from this file and prints them along with their sum.

- Output:

number = 308.2

number = 14.9

number = 7.4

number = 2.8

number = 3.9

Sum = 337.199999999999993

Solution 1

```
// Displays the first 5 numbers in the given file,  
// and displays their sum at the end.  
import java.io.*;    // for File, FileNotFoundException  
import java.util.Scanner;  
public class Echo {  
    public static void main(String[] args)  
        throws FileNotFoundException {  
        Scanner in = new Scanner(new File("numbers.dat"));  
        double sum = 0.0;  
        for (int i = 1; i <= 5; i++) {  
            double next = in.nextDouble();  
            System.out.println("number = " + next);  
            sum += next;  
        }  
        System.out.println("Sum = " + sum);  
    }  
}
```

Reading a whole file

- The preceding program is assumes you know how many values you want to read.
- How could we read in ALL of the numbers in the file, without knowing beforehand how many the file contains?
- The Scanner has useful methods for testing to see what the next input token will be.

Method Name	Description
<code>hasNext()</code>	whether any more tokens remain
<code>hasNextDouble()</code>	whether the next token can be interpreted as type <code>double</code>
<code>hasNextInt()</code>	whether the next token can be interpreted as type <code>int</code>
<code>hasNextLine()</code>	whether any more lines remain

Exercise 2

- Rewrite the previous program so that it reads the entire file.
- Output:
- number = 308.2
- number = 14.9
- number = 7.4
- number = 2.8
- number = 3.9
- number = 4.7
- number = -15.4
- number = 2.8
- Sum = 329.299999999999995

Solution 2

```
import java.io.*;    // for File, FileNotFoundException
import java.util.Scanner;
public class Echo2 {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNextDouble()) {
            double next = input.nextDouble();
            System.out.println("number = " + next);
            sum += next;
        }
        System.out.println("Sum = " + sum);
    }
}
```

Exercise 3

- Modify the preceding program again so that it will handle files that contain non-numeric tokens.
 - The program should skip any such tokens.
- For example, the program should produce the same output as before when given this input file:

308.2 hello

14.9 7.4 bad stuff 2.8

3.9 4.7 oops -15.4

:-) 2.8 @#*(\$&

Solution 3

```
import java.io.*;    // for File, FileNotFoundException
import java.util.Scanner;
public class Echo3 {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new File("numbers.dat"));
        double sum = 0.0;
        while (input.hasNext()) {
            if (input.hasNextDouble()) {
                double next = input.nextDouble();
                System.out.println("number = " + next);
                sum += next;
            } else {
                input.next();    // consume / throw away bad token
            }
        }
        System.out.println("Sum = " + sum);
    }
}
```

Line-based processing

- Reading a file line-by-line, general syntax:

```
Scanner input = new Scanner(new File("<file  
name>"));  
while (input.hasNextLine()) {  
    String line = input.nextLine();  
    //<process this line>;  
}
```

- The `nextLine` method returns the characters from the input cursor's current position to the nearest `\n` character.

Hello file

```
public class Main {  
    public static void main(String[] args) throws  
FileNotFoundException{  
        Scanner sc = new Scanner(new  
File("Hello.txt"));  
while(sc.hasNextLine()) {  
    System.out.println(sc.nextLine());  
}  
}  
}
```

Exercise 4

- Write a program that reads a text file and "quotes" it by putting a > in front of each line.

Input:

Hey Prof. Yates,

I would like to know more about files. Please explain them to me.

Sincerely,
Susie Q. Student

Output:

> Hey Prof. Yates,

> I would like to know more about files. Please explain them to me.

> Sincerely,
> Susie Q. Student

Solution 4

```
import java.io.*;
import java.util.*;
public class QuoteMessage {
    public static void main(String[] args)
        throws FileNotFoundException {
        Scanner input = new Scanner(new
File("message.txt"));
        while (input.hasNextLine()) {
            String line = input.nextLine();
            System.out.println(">" + line);
        }
    }
}
```

Exercise 5

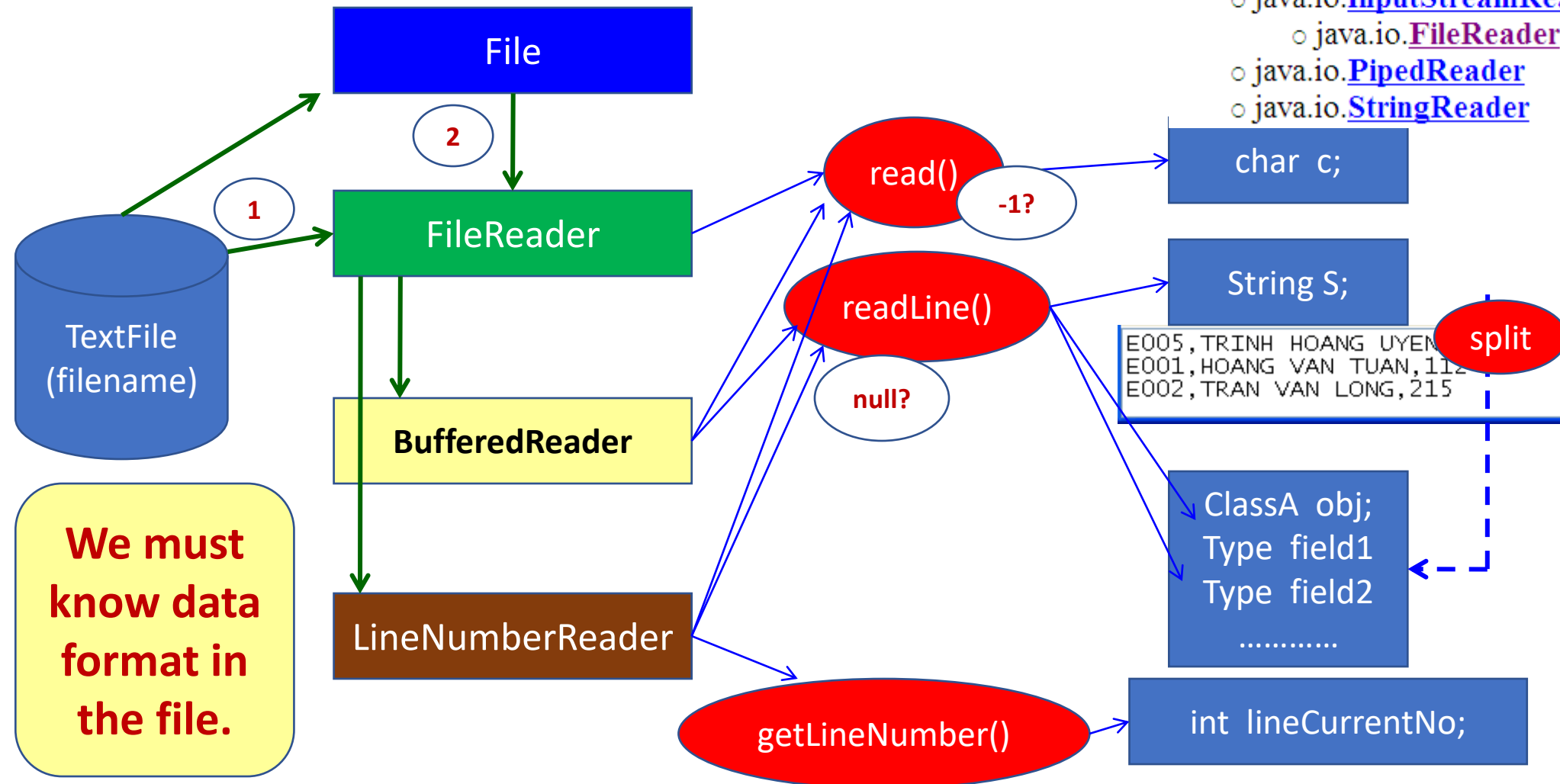
- Example file contents: (input)
123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jennifer 8.0 8.0 8.0 8.0 7.5
- Consider the task of computing the total hours worked for each person represented in the above file. (output)
Susan (ID#123) worked 31.4 hours (7.85 hours/day)
Brad (ID#456) worked 36.8 hours (7.36 hours/day)
Jennifer (ID#789) worked 39.5 hours (7.9 hours/day)

Access Text Files ...Character Streams

- `java.io.Reader` (implements `java.io.Closeable`, `java.lang.Readable`) (**abstract**)
 - `java.io.BufferedReader`
 - `java.io.LineNumberReader`
 - `java.io.CharArrayReader`
 - `java.io.FilterReader`
 - `java.io.PushbackReader`
 - `java.io.InputStreamReader`
 - `java.io.FileReader`
 - `java.io.PipedReader`
 - `java.io.StringReader`
- `java.io.Writer` (implements `java.lang.Appendable`, `java.io.Closeable`, `java.io.Flushable`) (**abstract**)
 - `java.io.BufferedWriter`
 - `java.io.CharArrayWriter`
 - `java.io.FilterWriter`
 - `java.io.OutputStreamWriter`
 - `java.io.FileWriter`
 - `java.io.PipedWriter`
 - `java.io.PrintWriter`
 - `java.io.StringWriter`

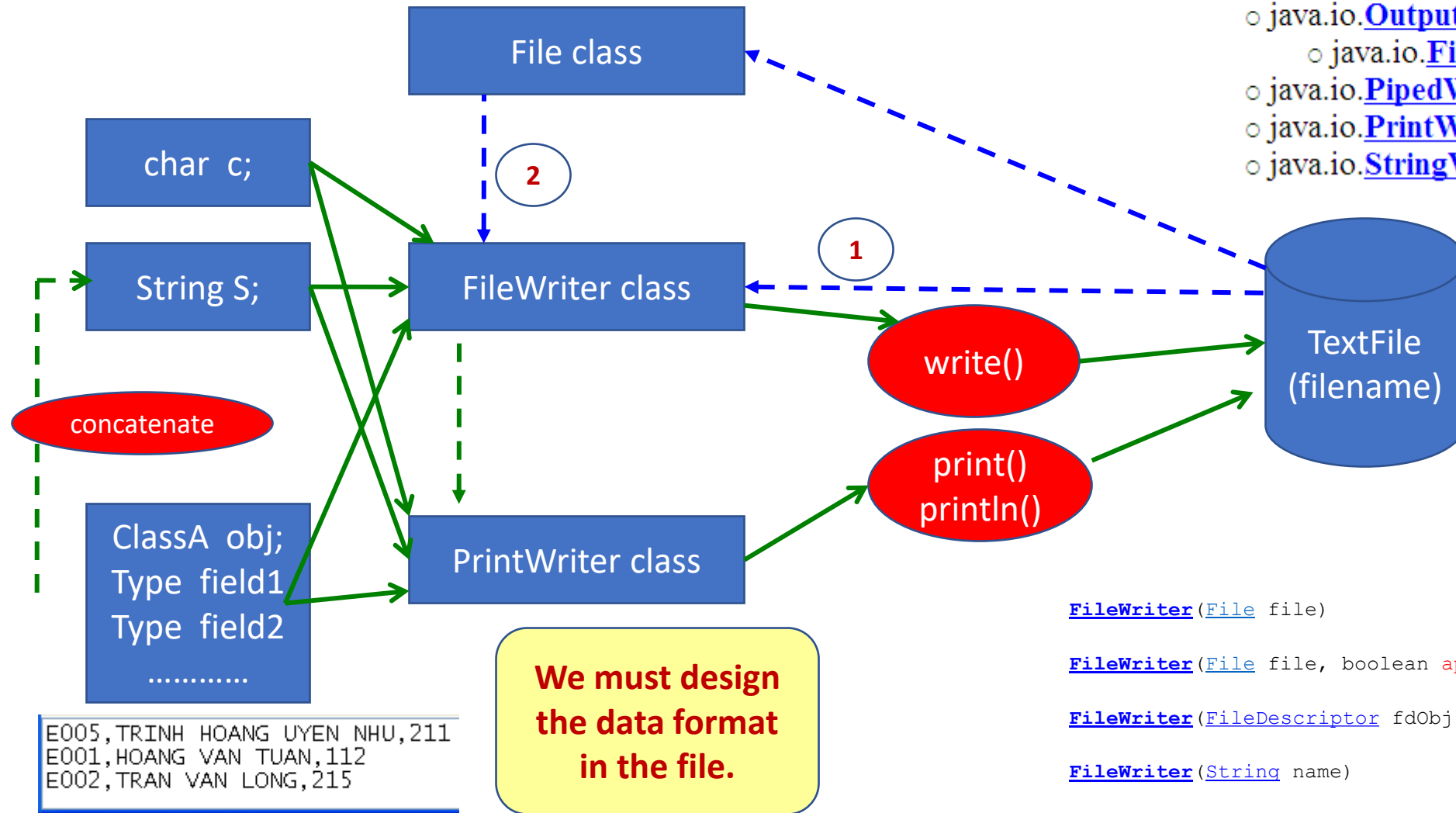
Reading Data

- java.io.Reader
 - java.io.BufferedReader
 - java.io.LineNumberReader
 - java.io.CharArrayReader
 - java.io.FilterReader
 - java.io.PushbackReader
 - java.io.InputStreamReader
 - java.io.FileReader
 - java.io.PipedReader
 - java.io.StringReader



Writing Data

- java.io.[Writer](#)
 - java.io.[BufferedWriter](#)
 - java.io.[CharArrayWriter](#)
 - java.io.[FilterWriter](#)
 - java.io.[OutputStreamWriter](#)
 - java.io.[FileWriter](#)
 - java.io.[PipedWriter](#)
 - java.io.[PrintWriter](#)
 - java.io.[StringWriter](#)



`FileWriter` (`File` file)

`FileWriter` (`File` file, boolean `append`)

`FileWriter` (`FileDescriptor` fdObj)

`FileWriter` (`String` name)

`FileWriter` (`String` name, boolean `append`)

Exercise 6

- Each employee details include: code, name, salary
- The text file, named employees.txt contains some initial employee details in the following line-by-line format
code, name, salary
- Write a Java program having a simple menu that allows users managing a list of employees. Functions are supported:
 - Adding new employee
 - Removing employee.
 - Promoting the salary of an employee.
 - Listing employee details.
 - Save the list to file
 - Quit

Design

Navigator

Members View

- Employee :: Comparable
 - Employee(String c, String n, int s)
 - compareTo(Object emp) : int
 - getCode() : String
 - getName() : String
 - getSalary() : int
 - print()
 - setCode(String code)
 - setName(String name)
 - setSalary(int salary)
 - code : String
 - name : String
 - salary : int

Navigator

Members View

- ManageProgram
 - main(String[] args)

employees.txt - Notepad

File Edit Format View Help

```
E005,TRINH HOANG UYEN NHU,211
E001,HOANG VAN TUAN,112
E002,TRAN VAN LONG,215
```

Output - Chapter09 (run)

```
EMPLOYEE MANAGER
1-Add new employee
2-Remove an employee
3-Promoting the employee's salary
4-Print the list
5-Save to files
6-Quit

Select 1..6: 4

EMPLOYEE LIST
-----
E001      HOANG VAN TUAN      112
E002      TRAN VAN LONG      215
E005      TRINH HOANG UYEN NHU  211

EMPLOYEE MANAGER
1-Add new employee
2-Remove an employee
3-Promoting the employee's salary
4-Print the list
5-Save to files
6-Quit

Select 1..6: |
```

```
11 // Add employees from a text file
12 public void AddFromFile(String fName) {
13     try {
14         File f= new File(fName); // checking the file
15         if (!f.exists()) return;
16         FileReader fr= new FileReader(f); // read()
17         BufferedReader bf= new BufferedReader(fr); // readLine()
18         String details ; // E001,Hoang Van Tuan,156
19         while ((details= bf.readLine()) !=null)
20         { // Splitting details into elements
21             StringTokenizer stk= new StringTokenizer(details, ",");
22             String code= stk.nextToken().toUpperCase();
23             String name= stk.nextToken().toUpperCase();
24             int salary = Integer.parseInt(stk.nextToken());
25             // Create an employee
26             Employee emp= new Employee(code, name, salary);
27             this.add(emp); // adding this employee to the list
28         }
29         bf.close(); fr.close();
30     }
31     catch(Exception e) {
32         System.out.println(e);
33     }
34 }
```

```
try{
    bf=new BufferedReader(new FileReader(f));
    String details="";
    while((details=bf.readLine())!=null){
        String[] s=details.split("\\\\,\\\\s*");
        String code=s[0];
        String name=s[1];
        int salary=Integer.parseInt(s[2]);
        Employee emp=new Employee(code,name,salary);
        this.add(emp);
    }
    bf.close();
}catch(FileNotFoundException e){
    System.out.println(e);
}catch(IOException e){
    System.out.println(e);
}catch(NumberFormatException e){
    System.out.println(e);
}
```

```
35 public void saveToFile (String fName){
36     if (this.size()==0) {
37         System.out.println("Empty list");
38         return;
39     }
40     try{
41         File f = new File(fName);
42         FileWriter fw = new FileWriter(f); // write()
43         PrintWriter pw = new PrintWriter(fw); // println()
44         for (Employee x:this) {
45             pw.println(x.getCode() + "," + x.getName() + "," + x.getSalary());
46         }
47         pw.close(); fw.close();
48     }
49     catch (Exception e){
50         System.out.println(e);
51     }
52 }
```


Read UTF-8 File content

UTF8 content is stored in compressed format → a character will be stored in 1 to 3 bytes.
Before reading UTF, decompressing is needed.

```
String content="";  
FileInputStream f = new FileInputStream(filename);  
InputStreamReader isr = new InputStreamReader(f, "UTF8");  
int ch;  
while ((ch = in.read()) > -1) content+=(char)ch;
```

For read bytes

For read a
unicode
character

Or
"UTF-8"

```
String content="", s;  
FileInputStream f = new FileInputStream(filename);  
InputStreamReader isr = new InputStreamReader(f, "UTF8");  
BufferedReader br = new BufferedReader (isr);  
while ( (s= br.readLine())!=null) content += s + "\n";
```

For read a
unicode
character or
string.

Access binary files

- Binary streams.
 - Low-level streams: reading/writing data byte-by-byte.
 - High-level stream: reading/writing general-format data (primitives – group of bytes that store typed-values)

The `java.io.RandomAccessFile` class

- It is used to read or modify data in a file that is compatible with the stream, or reader, or writer model
- Constructors

`RandomAccessFile(String file, String mode)`

`RandomAccessFile(File file, String mode)`

- Mode “r” to open the file for reading only
- Mode “rw” to open for both reading and writing
- Mode “rws” is same as rw and any changes to the file’s content or metadata (file attributes) take place immediately
- Mode “rwd” is same as rw, and changes to the file content, but not its metadata, take place immediately. Its metadata are updated only when the file is closed.

Example

A demo. for write data to a file then read data from the file

The try...catch statement must be used when accessing file – checked exception

```
Output - Chapter09 (run)
run:
Mắt nai
true
1234
37.456
Hoang an Huan
File length: 37
```

WRITE

READ

```
/* Use the RandomAccessFile class to write/read some data */
import java.io.*;

public class RandomAccessFileDemo {
    public static void main (String[] args){
        String fName="RandomAccessFileDemo.txt";
        String S1= "Mắt nai"; boolean b=true; int n= 1234;
        double x= 37.456; String S2="Hoang an Huan";
        byte[] ar= new byte[100]; // for reading ASCII characters
        try {
            RandomAccessFile f= new RandomAccessFile(fName, "rw");
            // Write data , positions: 0,1,2,3,4
            f.writeUTF(S1); f.writeBoolean(b); f.writeInt(n);
            f.writeDouble(x); f.writeBytes(S2);
            // Read data
            f.seek(0); // seek to BOF
            System.out.println(f.readUTF());
            System.out.println(f.readBoolean());
            System.out.println(f.readInt());
            System.out.println(f.readDouble());
            f.read(ar);
            System.out.println(new String (ar));
            System.out.println("File length: " + f.length());
            f.close();
        }
        catch (Exception e){
            System.out.println(e);
        }
    }
}
```

Binary Streams

C:\Programming\jdk1.6.0\docs\api\java\io\package-tree.html

- o java.io.[InputStream](#) (implements java.io.[Closeable](#))
 - o java.io.[ByteArrayInputStream](#)
 - o java.io.[FileInputStream](#)
 - o java.io.[FilterInputStream](#)
 - o java.io.[BufferedInputStream](#)
 - o java.io.[DataInputStream](#) (implements java.io.[DataInput](#))
 - o java.io.[LineNumberInputStream](#)
 - o java.io.[PushbackInputStream](#)
 - o java.io.[ObjectInputStream](#) (implements java.io.[ObjectInput](#), java.io.[ObjectStreamConstants](#))
 - o java.io.[PipedInputStream](#)
 - o java.io.[SequenceInputStream](#)
 - o java.io.[StringBufferInputStream](#)
-

C:\Programming\jdk1.6.0\docs\api\java\io\package-tree.html

- o java.io.[OutputStream](#) (implements java.io.[Closeable](#), java.io.[Flushable](#))
 - o java.io.[ByteArrayOutputStream](#)
 - o java.io.[FileOutputStream](#)
 - o java.io.[FilterOutputStream](#)
 - o java.io.[BufferedOutputStream](#)
 - o java.io.[DataOutputStream](#) (implements java.io.[DataOutput](#))
 - o java.io.[PrintStream](#) (implements java.lang.[Appendable](#), java.io.[Closeable](#))
 - o java.io.[ObjectOutputStream](#) (implements java.io.[ObjectOutput](#), java.io.[ObjectStreamConstants](#))
 - o java.io.[PipedOutputStream](#)

Low-Level Binary Stream Demo

```
public class LowLevelStreamDemo {
```

```
    /**...*/
```

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

```
        final char BLANK=32;
```

```
        final String fileName="LStream.txt";
```

```
        int[] a = {1, 2, 3, 4, 5};
```

```
        char n = '5';
```

```
        try {
```

```
            FileOutputStream os = new FileOutputStream(fileName);
```

```
            os.write(n);//begin writing
```

```
            os.write(BLANK);
```

```
            for(int i=0; i<5; i++){
```

```
                os.write(a[i]);
```

```
                os.write(BLANK);
```

```
            }
```

```
            for(int i=0; i<fileName.length(); i++){
```

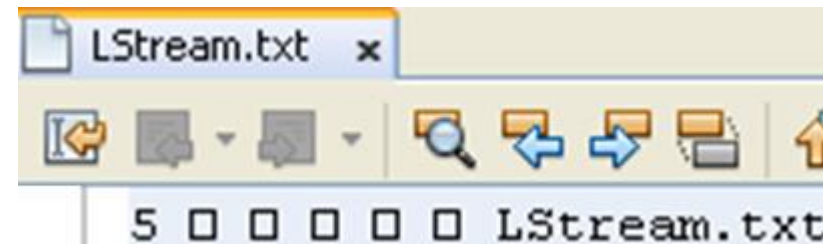
```
                os.write(fileName.charAt(i));
```

```
            }
```

```
            os.close();
```

These values can not be greater than 127 because only the lower bytes are written to the file.

Write
data to
file



We can not read these number in the file because of binary file. However, we can see characters.

Read
data
from
the file
then
print
them
out.

```
FileInputStream is = new FileInputStream(fileName);
int count = is.available();
System.out.println("The size of file is " + count + " bytes");
System.out.println("The content of file: ");
//read first char
byte[] bytes = new byte[1];
is.read(bytes);
System.out.print(new String(bytes));
//read blank
is.read(bytes);
System.out.print(new String(bytes));
//read int number
for(int i=0; i<5; i++){
    int tmp = is.read();
    is.read(bytes);
    System.out.print(tmp + new String(bytes));
}
bytes = new byte[11];
is.read(bytes);
System.out.println(new String(bytes));
is.close();
} catch(IOException e){
    e.printStackTrace();
}
}
```

Read a byte: '5'

Read the blank

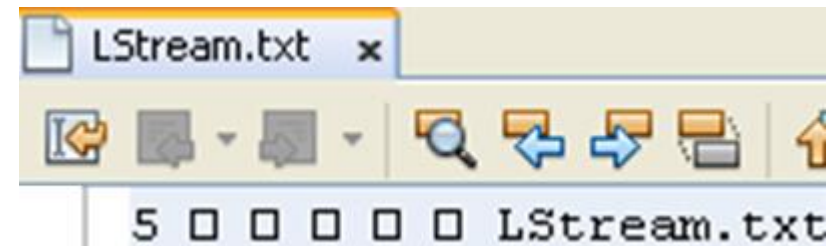
Read the blank

Read a number

Read filename stored at the end of the file

Convert array of characters to string
for printing them easier.

The size of file is 23 bytes
The content of file:
5 1 2 3 4 5 LStream.txt



High-Level Binary Stream (1)

- More often than not bytes to be read or written constitute higher-level information (int, String, ...)
- The most common of high-level streams extend from the super classes `FilterInputStream` and `FilterOutputStream`.
- Do not read/write from input/output devices such as files or sockets; rather, they read/write from other streams
 - `DataInputStream/ DataOutputStream`
 - Constructor argument: `InputStream/ OutputStream`
 - Common methods: `readXXX, writeXXX`
 - `BufferedInputStream/ BufferedOutputStream`: supports read/write in large blocks
 -

High-Level Binary Streams (2)

C:\Programming\jdk1.6.0\docs\api\java\io\package-tree.html

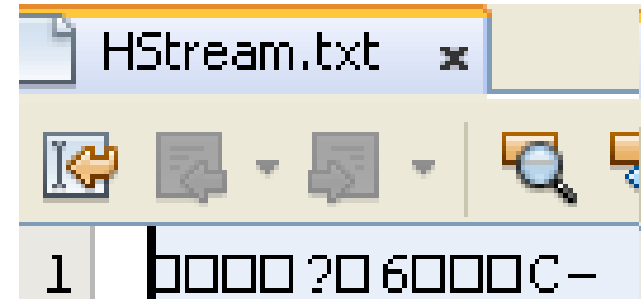
- java.io.[InputStream](#) (implements java.io.[Closeable](#))
 - java.io.[ByteArrayInputStream](#)
 - java.io.[FileInputStream](#)
 - java.io.[FilterInputStream](#)
 - java.io.[BufferedInputStream](#)
 - java.io.[DataInputStream](#) (implements java.io.[DataInput](#))
 - java.io.[LineNumberInputStream](#)
 - java.io.[PushbackInputStream](#)
 - java.io.[ObjectInputStream](#) (implements java.io.[ObjectInput](#), java.io.[ObjectStreamConstants](#))
 - java.io.[PipedInputStream](#)
 - java.io.[SequenceInputStream](#)
 - java.io.[StringBufferInputStream](#)

C:\Programming\jdk1.6.0\docs\api\java\io\package-tree.html

- java.io.[OutputStream](#) (implements java.io.[Closeable](#), java.io.[Flushable](#))
 - java.io.[ByteArrayOutputStream](#)
 - java.io.[FileOutputStream](#)
 - java.io.[FilterOutputStream](#)
 - java.io.[BufferedOutputStream](#)
 - java.io.[DataOutputStream](#) (implements java.io.[DataOutput](#))
 - java.io.[PrintStream](#) (implements java.lang.[Appendable](#), java.io.[Closeable](#))
 - java.io.[ObjectOutputStream](#) (implements java.io.[ObjectOutput](#), java.io.[ObjectStreamConstants](#))
 - java.io.[PipedOutputStream](#)

Example (1)

```
public class HighLevelStreamDemo {  
    /**...*/  
    public static void main(String[] args) {  
        final char BLANK=32;  
        final String fileName="HStream.txt";  
        int[] a ={1, 2, 3, 4, 5};  
        char n = '5';  
        try {  
            FileOutputStream os = new FileOutputStream(fileName);  
            DataOutputStream ds = new DataOutputStream(os);  
            ds.writeChar(n);//begin writing  
            ds.writeChar(BLANK);  
            for(int i=0; i<5; i++){  
                ds.writeInt(a[i]);  
                ds.writeChar(BLANK);  
            }  
            ds.writeUTF(fileName);  
            ds.close();  
            os.close();  
        }  
    }  
}
```



DataOutputStream
(int, string,...)

FileOutputStream
(byte)

File

A high-level
file access
includes some
low-level
access
(read an int
value includes
4 times of
read a byte)

Example (2)

```
FileInputStream is = new FileInputStream(fileName);
DataInputStream dis = new DataInputStream(is);
int count = dis.available();
System.out.println("The size of file is " + count + " bytes");
System.out.println("The content of file: ");
System.out.print(dis.readChar());
System.out.print(dis.readChar());
for(int i=0; i<5; i++){
    System.out.print(dis.readInt());
    System.out.print(dis.readChar());
}
System.out.println(dis.readUTF());
dis.close();
is.close();
} catch(IOException e){
    e.printStackTrace();
}
}
```

```
The size of file is 47 bytes
The content of file:
5 1 2 3 4 5 HStream.txt
```

Access Object Files

- 2 Object streams :Object Input stream, Object Output stream
- java.lang.[Object](#)
 - java.io.[InputStream](#) (implements java.io.[Closeable](#))
 - java.io.[ByteArrayInputStream](#)
 - java.io.[FileInputStream](#)
 - java.io.[FilterInputStream](#)
 - java.io.[ObjectInputStream](#) (implements java.io.[ObjectInput](#), java.io.[ObjectStreamConstants](#))
 - java.io.[OutputStream](#) (implements java.io.[Closeable](#), java.io.[Flushable](#))
 - java.io.[ByteArrayOutputStream](#)
 - java.io.[FileOutputStream](#)
 - java.io.[FilterOutputStream](#)
 - java.io.[ObjectOutputStream](#) (implements java.io.[ObjectOutput](#), java.io.[ObjectStreamConstants](#))

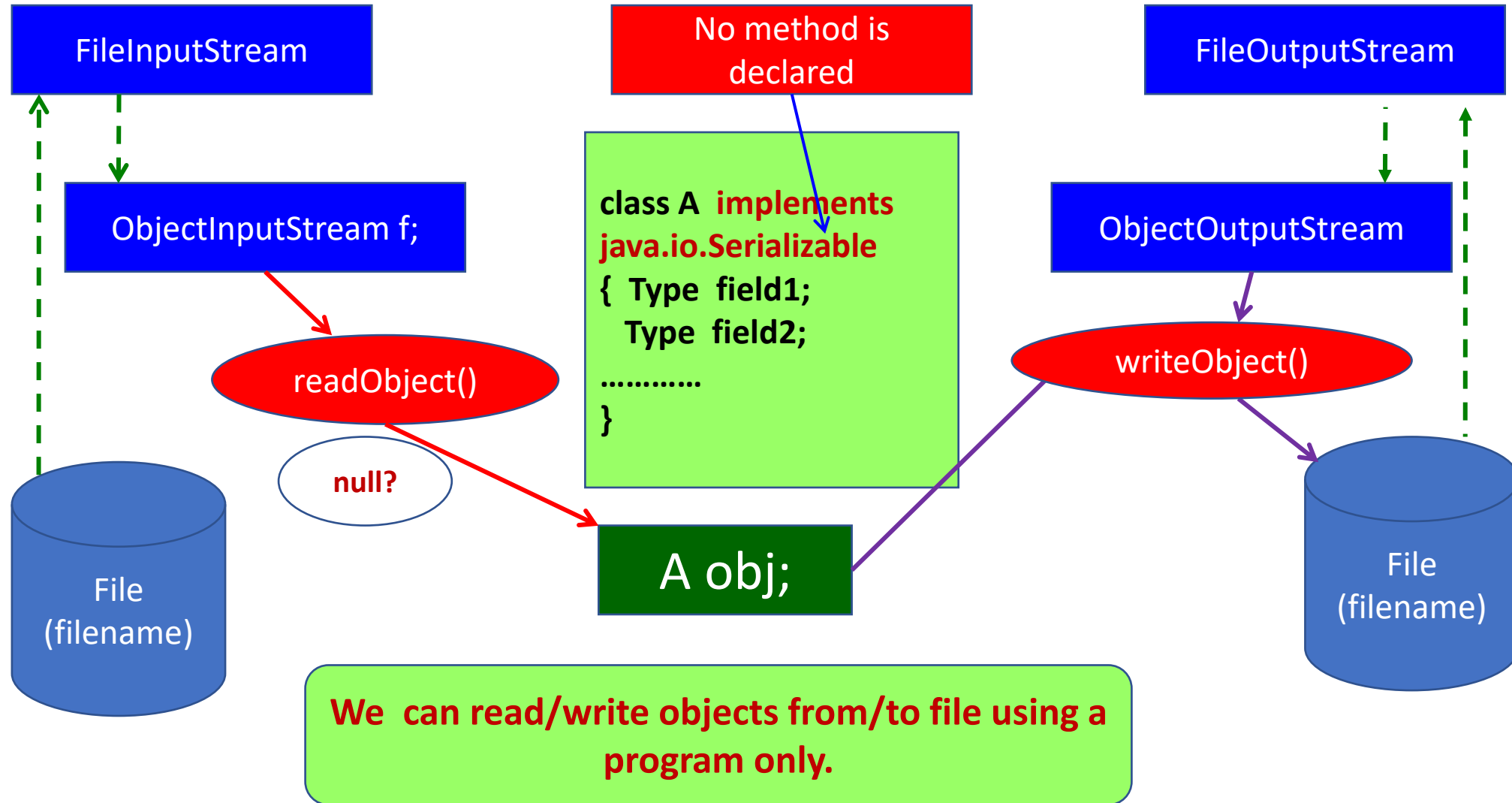
Serialization

- The process of writing an object is called *serialization*.
- Use `java.io.ObjectOutputStream` to serialize an object.
- It is only an object's data that is serialized, not its class definition.
- When an object output stream serializes an object that contains references to other object, every referenced object is serialized along with the original object.
- Not all data is written.
 - **static** fields are not
 - **transient** fields are also not serialized

De-serialization

- De-serialization is to convert a serialized representation into a replica of the original object.
- Use `java.io.ObjectInputStream` to deserialize an object.
- When an object is serialized, it will probably be deserialized by a different JVM.
- Any JVM that tries to deserialize an object must have access to that object's class definition.

Access Object Files...: How to?



Case study

Problem

- Student <code, name>
- Write a Java program that allows user:
 - View students in the file students.dat
 - Append list of students to the file
- Read/ Write students as binary objects from/to the file.

Student.java

```
public class Student implements Serializable{  
    private int code;  
    private String name;  
    public Student() {}  
    public Student(int code, String name) {  
        this.code = code;  
        this.name = name;  
    }  
    public String toString() {  
        return code+"\t"+name;  
    }  
}
```

```
import java.util.ArrayList;
import java.util.List;
//doc ra
import java.io.ObjectInputStream;
import java.io.FileInputStream;
//ghi vao
import java.io.ObjectOutputStream;
import java.io.FileOutputStream;
import java.io.IOException;
```

```
public class IOFile {
    //doc ra
    public static <T> List<T> read(String file){
        List<T> list=new ArrayList<>();
        try{
            ObjectInputStream o=new ObjectInputStream(
                new FileInputStream(file));
            list=(List<T>)o.readObject();
            o.close();
        }catch(IOException e){
            System.out.println(e);
        }catch(ClassNotFoundException e){
            System.out.println(e);
        }
        return list;
    }
}
```

```
//ghi vao
public static <T> void
    write(String file,List<T> arr){
    try{
        ObjectOutputStream o=
            new ObjectOutputStream(
                new FileOutputStream(file));
        o.writeObject(arr);
        o.close();
    }catch(IOException e){
        System.out.println(e);
    }
}
```

```
public static void main(String[] args) {
    List<Student> lists=new ArrayList<>();
    lists.add(new Student(1,"To An An"));
    lists.add(new Student(2,"Tuan Binh"));
    lists.add(new Student(3,"Vu Thi Teo"));
    lists.add(new Student(4,"Dang Nhat Minh"));
    String fs="src/io/student.dat";
    IOFile.write(fs, lists);
}
```

Summary

- Exception Handling
- Multiple Handlers
- Code Finalization and Cleaning Up (finally block)
- Custom Exception Classes
- Files and Variables
- Distinguishing Text, UTF, and Unicode
- How to access directories and files?
- How to access text files.
- How to access binary files
- How to read/write objects from/to files