JAVA BACKEND DEVELOPMENT PROGRAM

Exceptions & Java 8

OUTLINE

- Types of Exception
- Exception handling
- Custom Exception
- Java 8 Features
- Java I/O
- Serialization

EXCEPTIONS IN JAVA

EXCEPTION

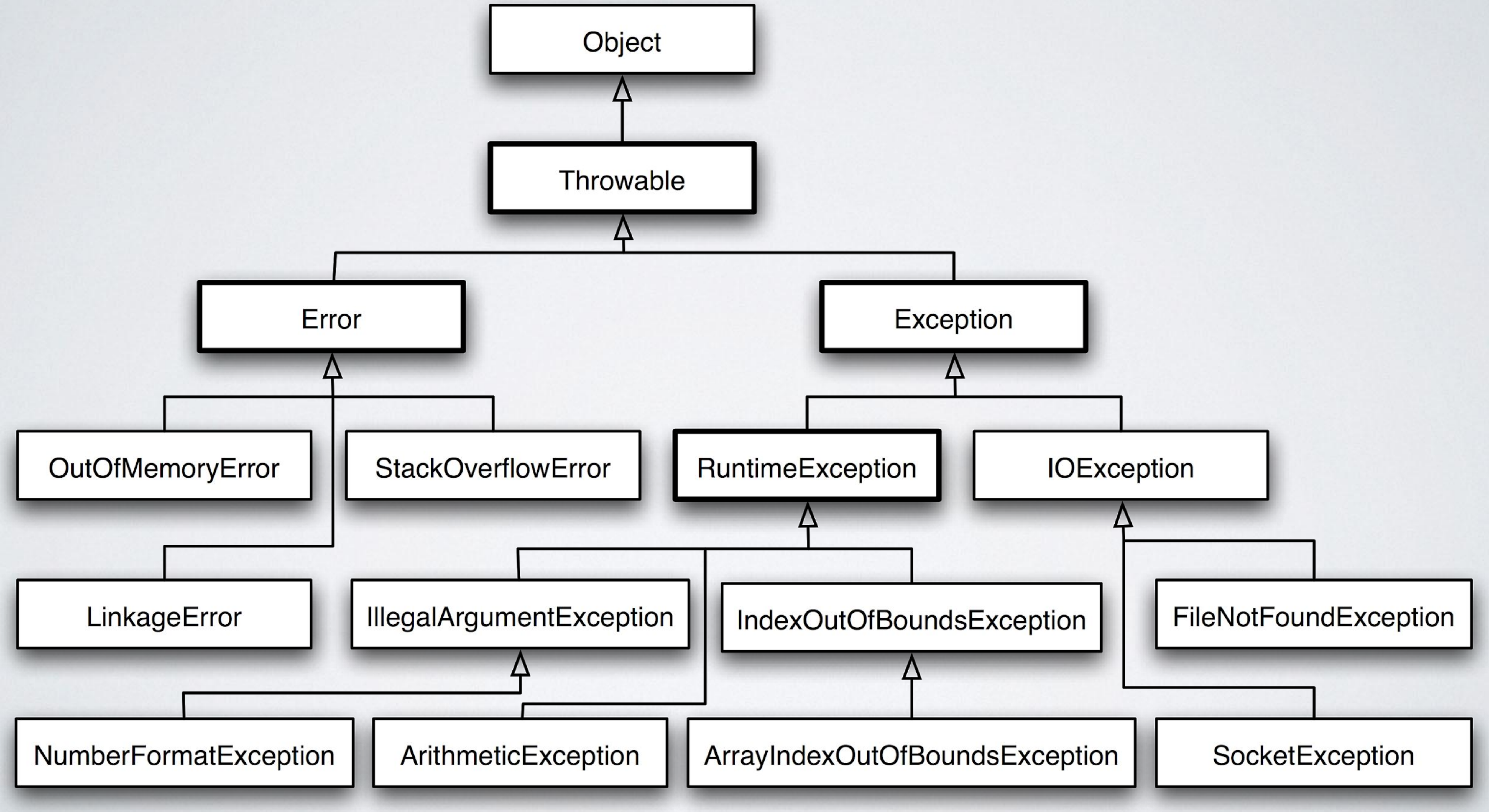
- An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions.
- Exceptions can arise due to a number of situations.
  - Trying to access the 11th element of an array when the array contains of only 10 elements. (*ArrayIndexOutOfBoundsException*)
  - Division by zero (*ArithmeticException*)
  - Accessing a file which is not (*FileNotFoundException*)
  - Failure of I/O operations (*IOException*)

EXCEPTION VS ERROR

- **Exception** — Exception indicates conditions that a reasonable application might try to catch.
  - *ArithmeticException* — do not divide a number by 0

- **Error** — An Error indicates serious problem that a reasonable application should not try to catch.
  - Mostly Error is thrown by JVM in a scenario which is fatal and there is no way for the application program to recover from that error. For instance *OutOfMemoryError*

EXCEPTION HIERARCHY



TYPES OF EXCEPTION

- Checked exception
  - checked by the compiler at compile time
  - IOException, SQLException, ClassNotFoundException
- Unchecked exception
  - checked by the JVM at run time
  - ArrayIndexOutOfBoundsException, NullPointerException

WHY TWO TYPE OF EXCEPTIONS

- Checked Exception — Those who call a method must know about the exception so that they can handle properly.
- Unchecked Exception — Runtime Exception may happen everywhere (very common). Adding it to the method declaration will reduce the program clarity.
- General Rule — If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

EXCEPTION HANDLING

- try-catch
- throw/throws
- finally

TRY-CATCH

- try/catch block can be placed within any method that you feel can throw exceptions
- All the statements to be tried for exceptions are put in a try block
- catch block is used to catch any exception raised from the try block
- If exception occurs in any statement in the try block control immediately passes to the corresponding catch block.

TRY-CATCH

```
static void method2()
{
        System.out.println(“IN Method 2, Calling Method 3”);
        try{
                method3(); }
        catch(ArithmeticException ae)
        {
                System.out.println (“Arithmetic Exception Handled: “ +ae);
        }
        catch(Exception e)
        {
                System.out.println(“Exception Handled”);
        }
System.out.println(“Returned from method 3”);
}
```

TRY-CATCH

- Order of catch is important
    - catch having super class types should be defined later than the catch clauses with subclass types
- Nested try-catch block is allowed.
- Multi catch is available since Java 7:
    - eg. catch (Exception1 | Exception2 | Exception3)

ORDER OF CATCH

- Always put smaller exceptions in front of bigger exceptions
- Don’t put big basket in the front; you are going to catch everything
- Don’t put small basket in the end; you will not catch anything

FINALLY

- **finally** block is executed in all circumstances
  - if the exception occurs or
  - it is normal return (using return keyword) from methods
- Mandatory to execute statements like related to release of resources, etc. can be put in a **finally** block

THROW

- Used to explicitly throw an exception
- Useful when we want to throw a user-defined exception.
- The syntax for *throw* keyword is as follows:
    - throw new Throwable Instance
    - eg. throw new NullPointerException();

THROWS

- Added to the method signature to let the caller know about what exception the called method can throw

- It is the responsibility of the caller to either handle the exception (using try...catch mechanism) or it can also pass the exception (by specifying throws clause in its method declaration)

- If all the methods in a program pass the exception to their callers (including main( )), then ultimately the exception passes to the default exception handler

CUSTOM EXCEPTION

- A class , which is a subclass of Exception or RuntimeException
- Provide constructor as needed
- Use throw keyword to throw exception whenever you want to raise the exception
- Can be either checked or unchecked exception
  - Extend Exception class if you want to create checked exception
  - Extend RuntimeException class if you want to create unchecked exception

JAVA 8
NEW FEATURES

Java 8

Java 8 new features:

1. Optional Class

2. Functional Interface and Lambda Expressions

3. Default and static methods for interfaces

4. Stream API for Bulk Data Operations on Collections

5. forEach() method in Iterable interface

   (……. and more!)

OPTIONAL CLASS

Optional Class

- Optional is a container object used to contain not-null objects. Optional object is used to represent null with absent value.
- This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values.
- Useful for circumventing handling of NullPointerException

Optional Class

- Useful Methods

static <T> Optional<T> empty()	Returns an empty Optional instance.
static <T> Optional<T> of(T value)	Returns an Optional with the specified present non-null value.
static <T> Optional<T> ofNullable(T value)	Returns an Optional describing the specified value, if non-null, otherwise returns an empty Optional.
void ifPresent(Consumer<? super T> consumer)	If a value is present, it invokes the specified consumer with the value, otherwise does nothing.
boolean isPresent()	Returns true if there is a value present, otherwise false.
T orElse(T other)	Returns the value if present, otherwise returns other.
T orElseGet(Supplier<? extends T> other)	Returns the value if present, otherwise invokes other and returns the result of that invocation.
Optional<T> filter(Predicate<? super <T> predicate)	If a value is present and the value matches a given predicate, it returns an Optional describing the value, otherwise returns an empty Optional.

https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

FUNCTIONAL INTERFACES

CONSUMER INTERFACE

- Consumer is a functional interface which represents an operation that accepts a single input argument and returns no result

- Methods:
    - accept()
    - andThen()

SUPPLIER INTERFACE

- Supplier is a functional interface which represents an operation that takes no argument and returns a result

- Methods:
    - get()

FUNCTION INTERFACE

- Function is a functional interface which represents an operation that takes an argument and returns a result
- The argument and result can be different types

- Methods:
    - apply()
    - andThen()
    - compose()
    - identity()

PREDICATE INTERFACE

- Predicate is a functional interface which represents an operation that takes an argument and returns a boolean result

- Methods:
    - test()
    - and()
    - or()
    - negate()

STREAM API

STREAM API

- Introduced in Java 8, the Stream API is used to process collections of objects.
- A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

STREAM API

- Main Features:
  - A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
  - Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
  - Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined. Terminal operations mark the end of the stream and return the result.

Intermediate Operations VS Terminal Operations

- Main Difference:
  - intermediate operations return a stream as a result and terminal operations return non-stream values like primitive or object or collection or may not return anything.
  - Intermediate operations are lazily loaded. When you call intermediate operations, they are actually not executed. They are just stored in the memory and executed when the terminal operation is called on the stream.

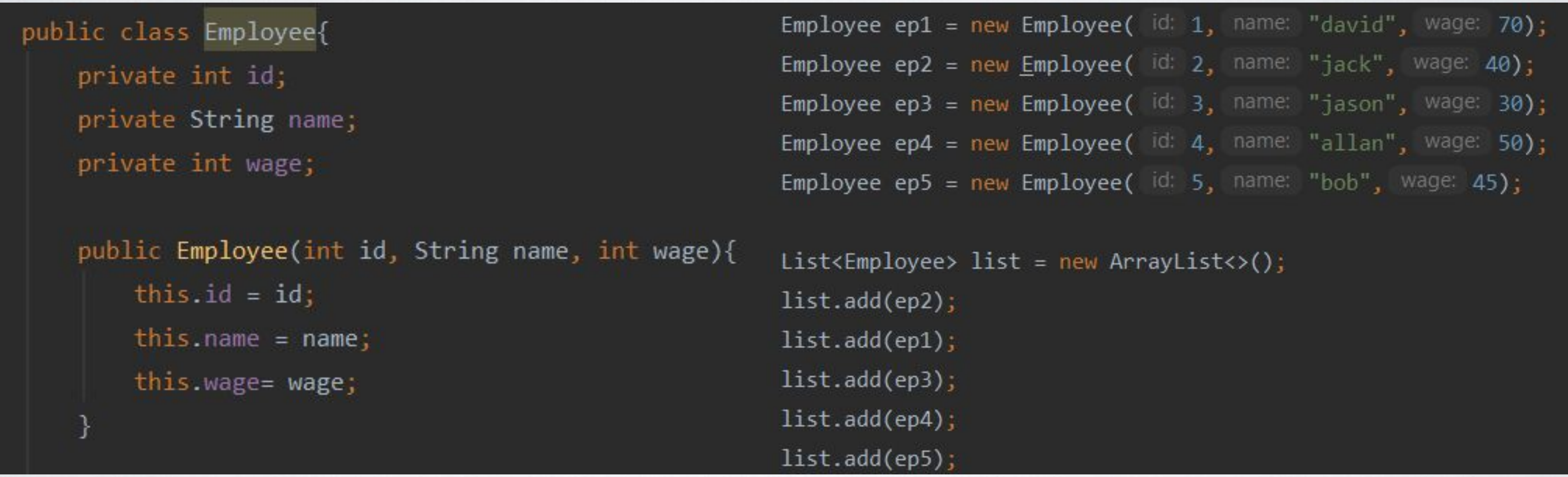
Intermediate Operations VS Terminal Operations

Intermediate Operations	map(), filter(), distinct(), sorted(), limit(), skip()
Terminal Operations	forEach(), toArray(), reduce(), collect(), min(), max(), count(), anyMatch(), allMatch(), noneMatch(), findFirst(), findAny()

https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html

STREAM API EXAMPLE

Consider Following Question:



```java
public class Employee{
    private int id;
    private String name;
    private int wage;

    public Employee(int id, String name, int wage){
        this.id = id;
        this.name = name;
        this.wage= wage;
    }
```

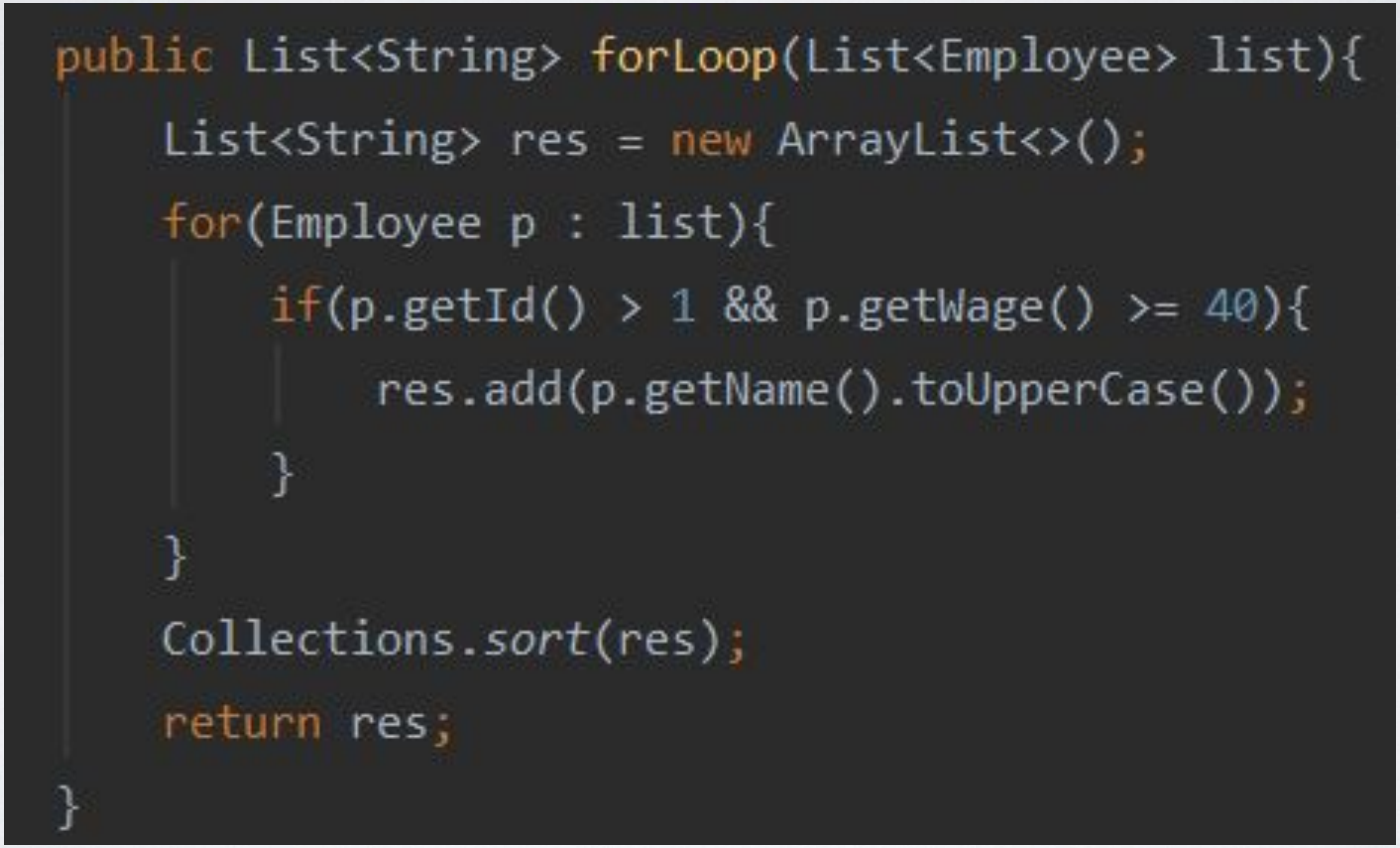
```java
Employee ep1 = new Employee( id: 1, name: "david", wage: 70);
Employee ep2 = new Employee( id: 2, name: "jack", wage: 40);
Employee ep3 = new Employee( id: 3, name: "jason", wage: 30);
Employee ep4 = new Employee( id: 4, name: "allan", wage: 50);
Employee ep5 = new Employee( id: 5, name: "bob", wage: 45);

List<Employee> list = new ArrayList<>();
list.add(ep2);
list.add(ep1);
list.add(ep3);
list.add(ep4);
list.add(ep5);
```

return a String List, which contains the names of the employees whose

- ID >1
- Wage >= 40

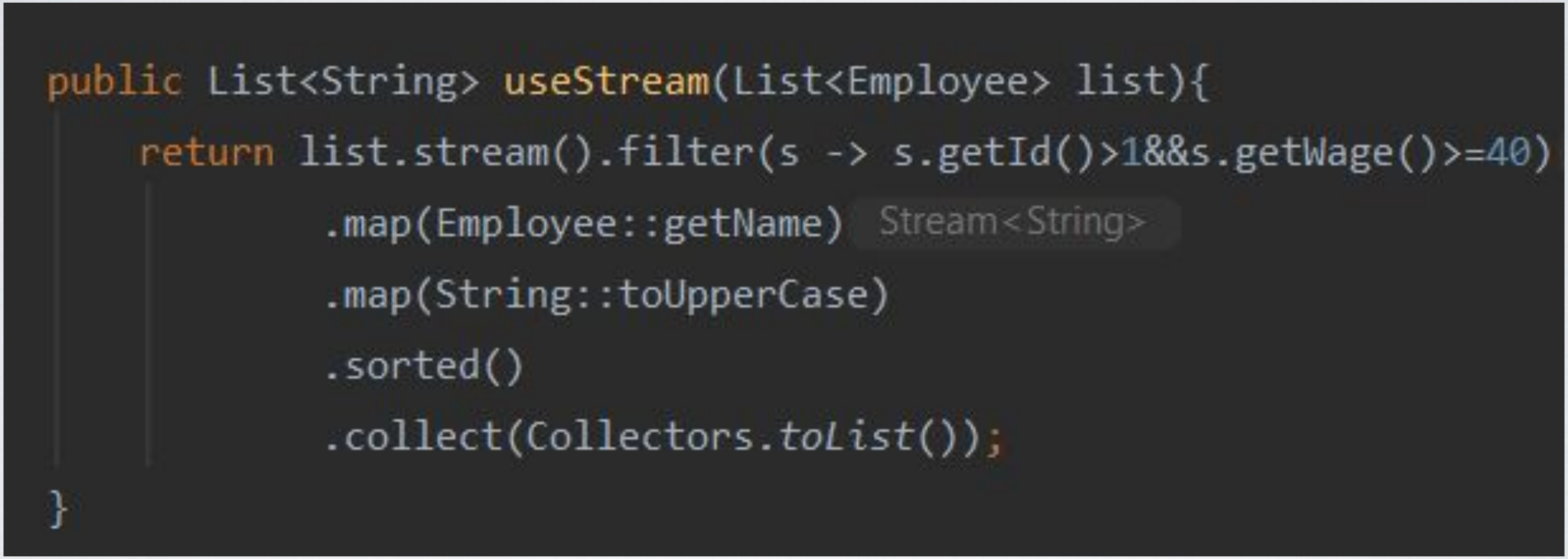
And the result should be in uppercase and in Alphabetical order

For Loop Solution



```java
public List<String> forLoop(List<Employee> list){
    List<String> res = new ArrayList<>();
    for(Employee p : list){
        if(p.getId() > 1 && p.getWage() >= 40){
            res.add(p.getName().toUpperCase());
        }
    }
    Collections.sort(res);
    return res;
}
```

Stream Solution



```
public List<String> useStream(List<Employee> list){
    return list.stream().filter(s -> s.getId()>1&&s.getWage()>=40)
            .map(Employee::getName) Stream<String>
            .map(String::toUpperCase)
            .sorted()
            .collect(Collectors.toList());
}
```

JAVA I/O

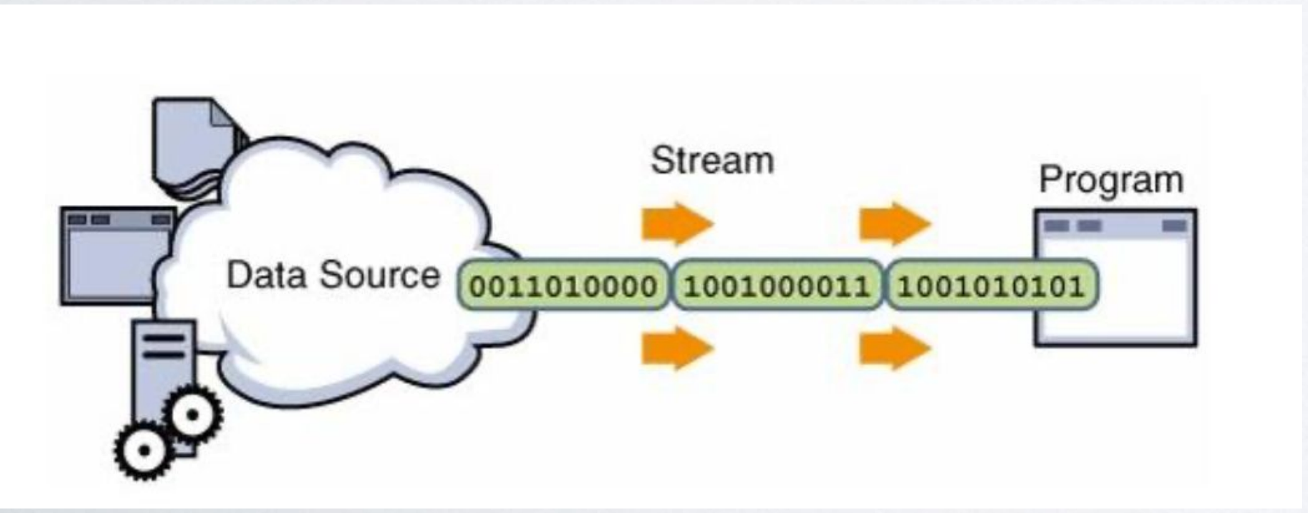
JAVA I/O

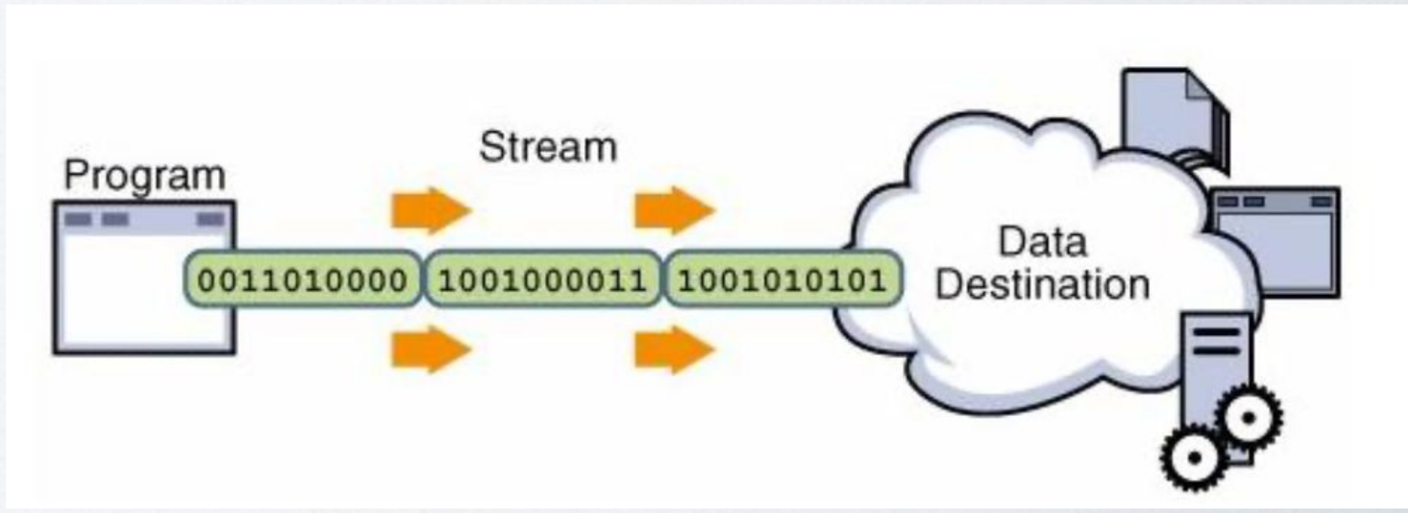
- Java I/O (Input and Output) is used to process the input and produce the output (read and write data). Most application need to process some data and produce some output based on the input.

- The java.io package contains all the classes required for input and output operations

STREAMS

- Java uses the concept of STREAM to make I/O operation fast.
- A stream is a conceptually endless flow of data. We can either read from a stream or write to a stream.
- A stream is connected to a data source or a data destination





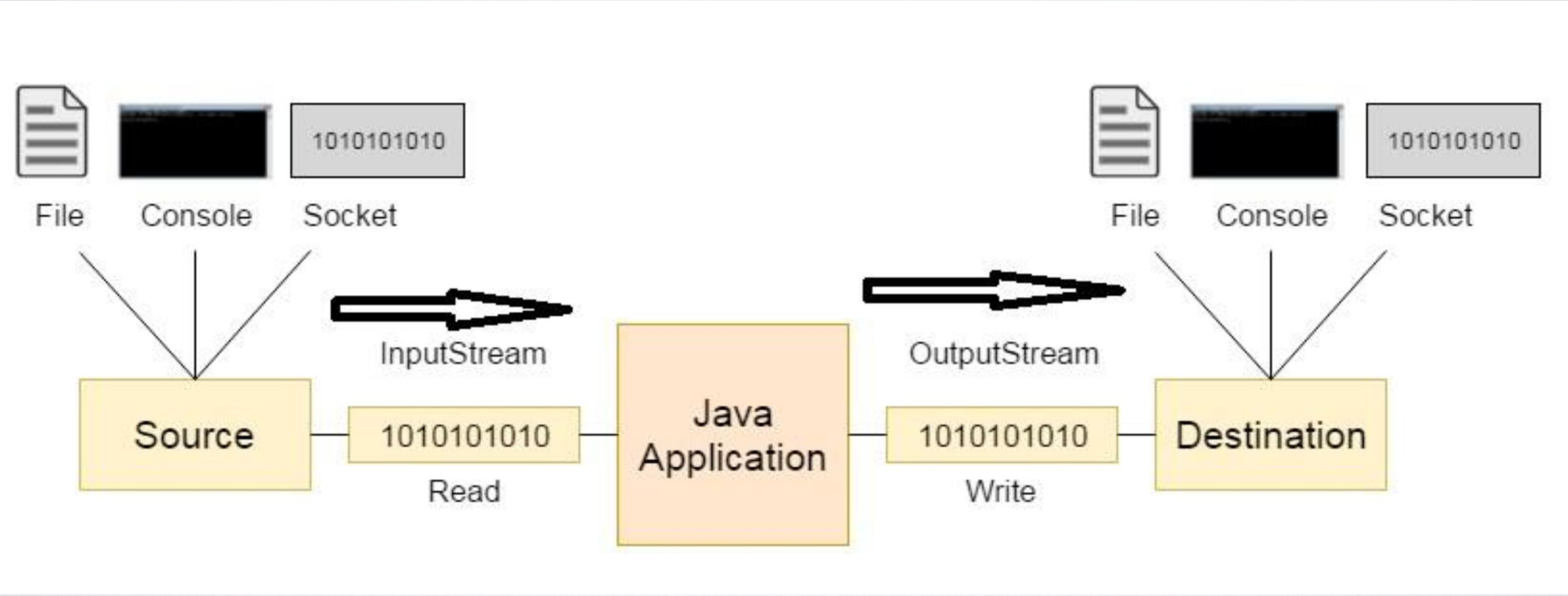
STREAMS

- In Java, stream can either be byte based (reading and writing bytes) or character based (reading and writing character)
- Stream does not have concept of an index (like array) , nor can we typically move forward and backward (like array)
- It is just a flow of data.

STREAM DATA TYPE

- Stream has two major data type: byte and character.
- Byte Stream:
  - It provides a convenient means for handling input and output of byte - used when working with bytes or other binary objects
  - The streams that are byte based are typically called "name + stream", e.g.: **InputStream, OutputStream**
- Character Stream :
  - It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.– used when working with characters or strings.
  - The streams that are character based are typically called "name + Reader" and "name + Writer". e.g.: **InputStreamReader, OutputStreamWriter**

INPUT AND OUTPUT BYTE STREAMS



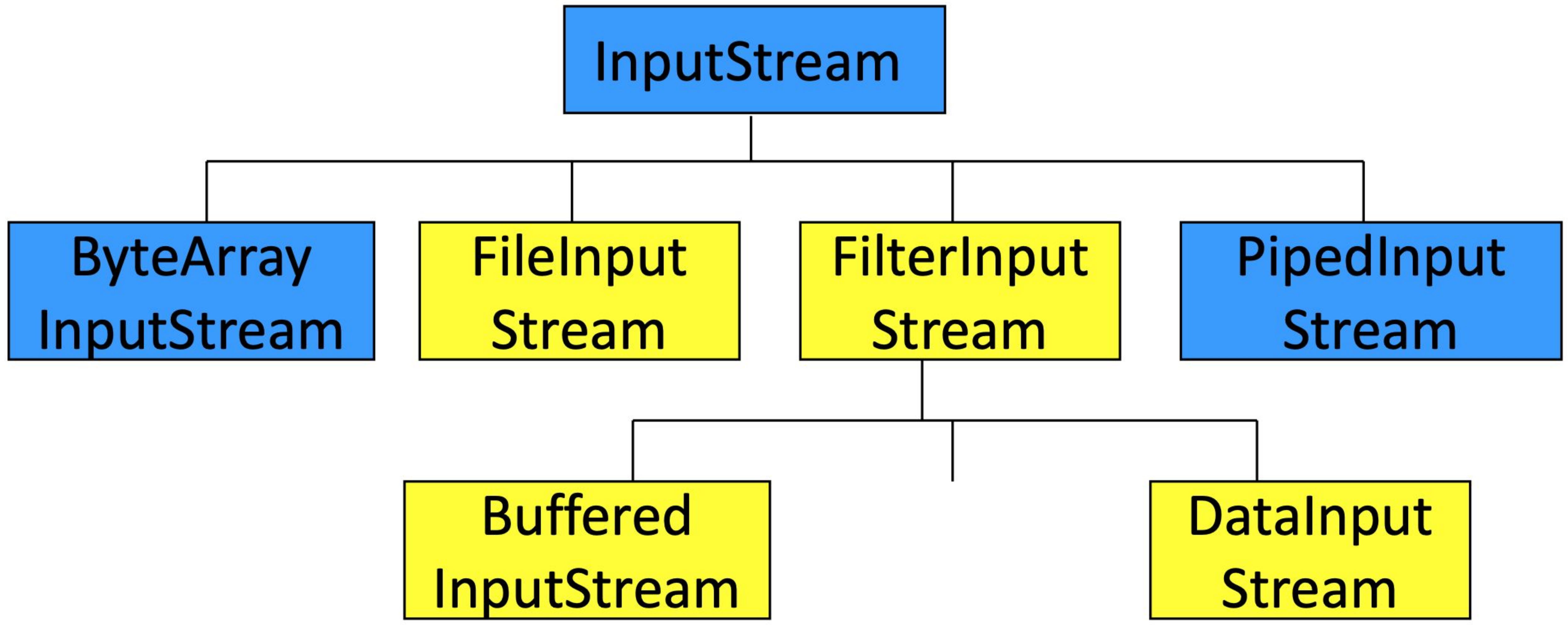
INPUTSTREAM

- Most of the methods in this class will throw an **IOException** when an I/O error occurs
- Some methods:
  - read() — read a byte from input stream
  - read(byte[] b) — read a byte array from input into b
  - read(byte[]b, int n, int m) — read m bytes into b from nth byte
  - close() — closes the input stream
- Read() method returns actual number of bytes that were successfully read or -1 if end of the file is reached.

OUTPUTSTREAM

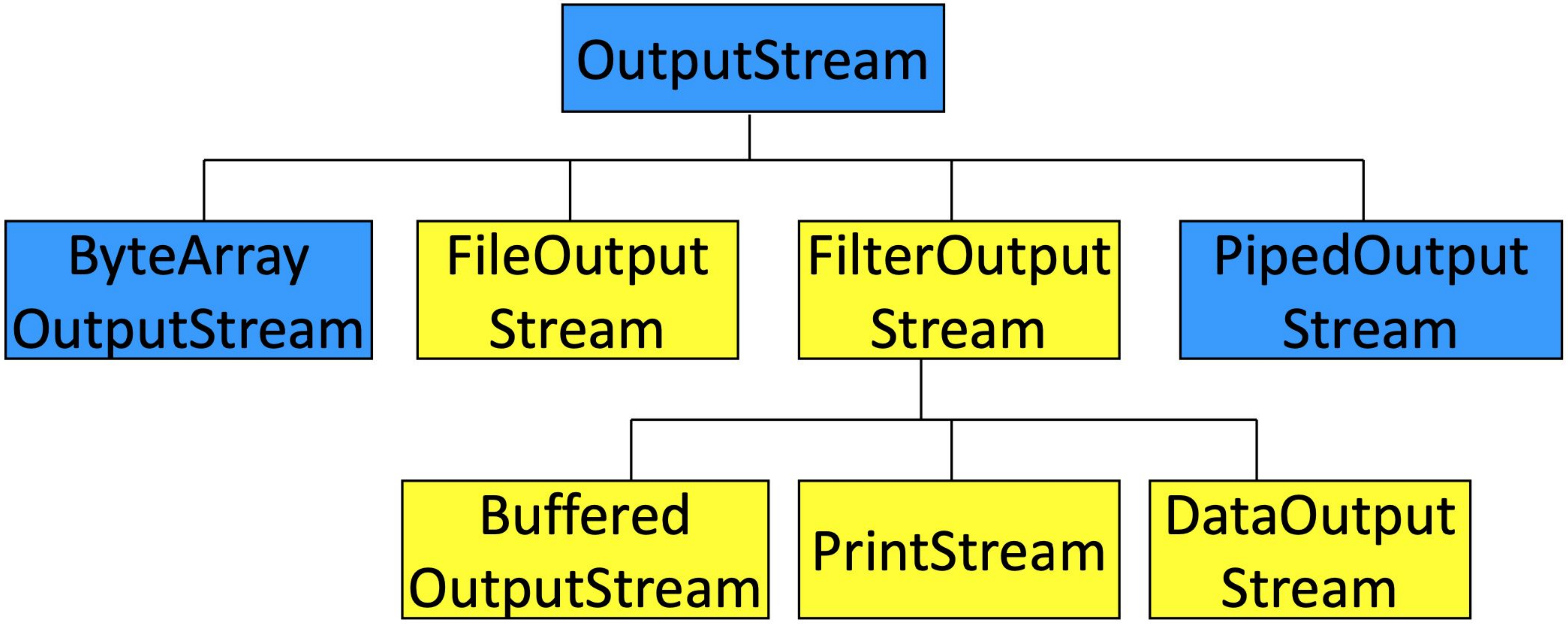
- OutputStream is an abstract class that defines streaming byte output.
- Most of the methods in this class return void and throw an **IOException** in the case of I/O errors.
- OutputStream has following methods:
  - write() — write a byte to output stream
  - write(byte[] b) — write all bytes in b into output stream
  - write(byte[] b, int n, int m) — write m bytes from array b from n’th
  - close() — Closes the output stream
  - flush() — flushes the output stream

INPUTSTREAM



https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/InputStream.html

OUTPUTSTREAM



https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/OutputStream.html

BUFFER

- It is a region of a physical memory storage used to temporarily store data while it is being moved from one place to another.

- This physical memory storage would be RAM (Random-access memory) in most cases.

- You might be using it without noticing. It doesn’t has to be in the name. BufferedWriter and BufferedReader are using buffers.

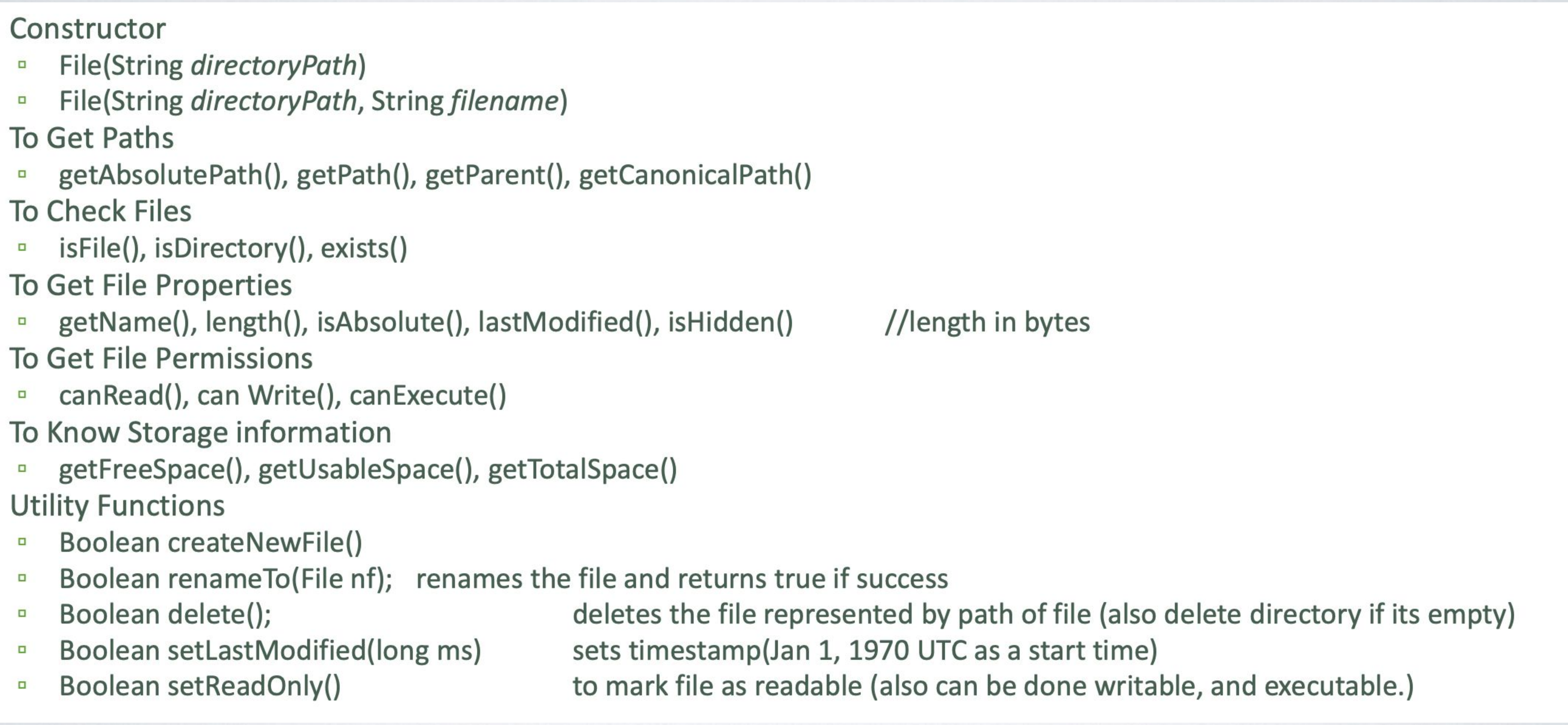
FILEREADER AND FILEWRITER

- The FileReader class creates a Reader that you can use to read the contents of a file.
- Reading from file
    - FileReader(String filePath)
    - FileReader(File fileObj)
- Writing to File
    - FileWriter(String fileName)
    - FileWriter(String fileName, boolean append)
    - If append is true, then the file is appended not overwritten

FILE (JAVA.IO.FILE)

- Most of the classes defined by java.io operate on streams, the File class does not.
- File deals directly with files and the file system. That is, the File class does not specify how information is retrieved from or stored in files; it describes the properties of a file itself.
- File object is used to obtain or manipulate the information associated with a disk file, such as the permissions, time, date, and directory path, and to navigate subdirectory hierarchies.
- We should use standard stream input/output classes to direct access for reading and writing file data

IO METHODS ON FILE



Constructor

- File(String *directoryPath*)
- File(String *directoryPath*, String *filename*)

To Get Paths

- getAbsolutePath(), getPath(), getParent(), getCanonicalPath()

To Check Files

- isFile(), isDirectory(), exists()

To Get File Properties

- getName(), length(), isAbsolute(), lastModified(), isHidden() //length in bytes

To Get File Permissions

- canRead(), can Write(), canExecute()

To Know Storage information

- getFreeSpace(), getUsableSpace(), getTotalSpace()

Utility Functions

- Boolean createNewFile()
- Boolean renameTo(File nf); renames the file and returns true if success
- Boolean delete(); deletes the file represented by path of file (also delete directory if its empty)
- Boolean setLastModified(long ms) sets timestamp(Jan 1, 1970 UTC as a start time)
- Boolean setReadOnly() to mark file as readable (also can be done writable, and executable.)

FILE IN JAVA

```java
public static void main(String[] args) {
    //accept file name or directory name through command line args
    String fname =args[0];

    //pass the filename or directory name to File object
    File f = new File(fname);

    //apply File class methods on File object
    System.out.println("File name :"+f.getName());
    System.out.println("Path: "+f.getPath());
    System.out.println("Absolute path:" +f.getAbsolutePath());
    System.out.println("Parent:"+f.getParent());
    System.out.println("Exists :"+f.exists());
    if(f.exists())
    {
        System.out.println("Is writeable:"+f.canWrite());
        System.out.println("Is readable"+f.canRead());
        System.out.println("Is a directory:"+f.isDirectory());
        System.out.println("File Size in bytes "+f.length());
    }
}
```

SERIALIZATION

SERIALIZATION

- It is a mechanism provided in Java.
- It allows Java to convert object information into bytes that includes the object's data and type. The process is called **serialization**.
- Once serialized, object will be stored in file as byte. This file can be read by other input streams and convert back into Java Objects. This process is called **deserialization**.
- Why is this important? The process is platform independent, meaning an object can be serialized on one platform and deserialized in another.
- In Java we have *ObjectInputStream* and *ObjectOutputStream*. These are high level streams that contains the methods for serializing and deserializing an object.

SERIALIZABLE

- A marker/tagging interface is an interface that has no methods or constants inside it. It provides run-time type information about objects, so the compiler and JVM have additional information about the object. e.g. *Serializable marker interface.*
- Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized.

SERIALIZATION

- Think of serialization as flatting an object
- If you are serializing an object that has primitives as instance variables, it will be easy
- However think about how would you serialize a tree structure? (You have to make sure when you deserialize the stored data, you can get the correct tree back.)
- Serialization is all or nothing. You can have a graph of objects, but if one of them fails at serialization, nothing will be serialized.

SERIALIZATION

- What if we do not want to serialize a variable?
- Use Keyword: transient
- Will static variable be serialized?

DESERIALIZATION

FileInputStream —> ObjectInputStream —> objects.

Objects are of Object type. It’s developers job to cast them to a specific type.

JVM will load the class and restore the objects without calling their constructors. Constructors will be called only for those of non-serializable classes.

Transient variables are given null or default values (0, false, etc.) for primitives.

WHAT CAN HURT DESERIALIZATION

- Deleting an instance variable
- Changing the declared type of an instance variable
- Changing a non-transient instance variable to transient
- Moving a class up or down the inheritance hierarchy
- Removing “implements Serializable
- Changing an instance variable to static

SERIALVERSIONUID

- Built-in static variable called serialVersionUID
- If you change your class, JVM will think the class is different now and will throw exception
- You can manually write the serialVersionUID to signal the JVM it’s OK to keep deserializing
- In the real world, very few developer does this. Usually new classes are created and new tables are created.

QUESTIONS?