Εργαστήριο #3

12/05/2021

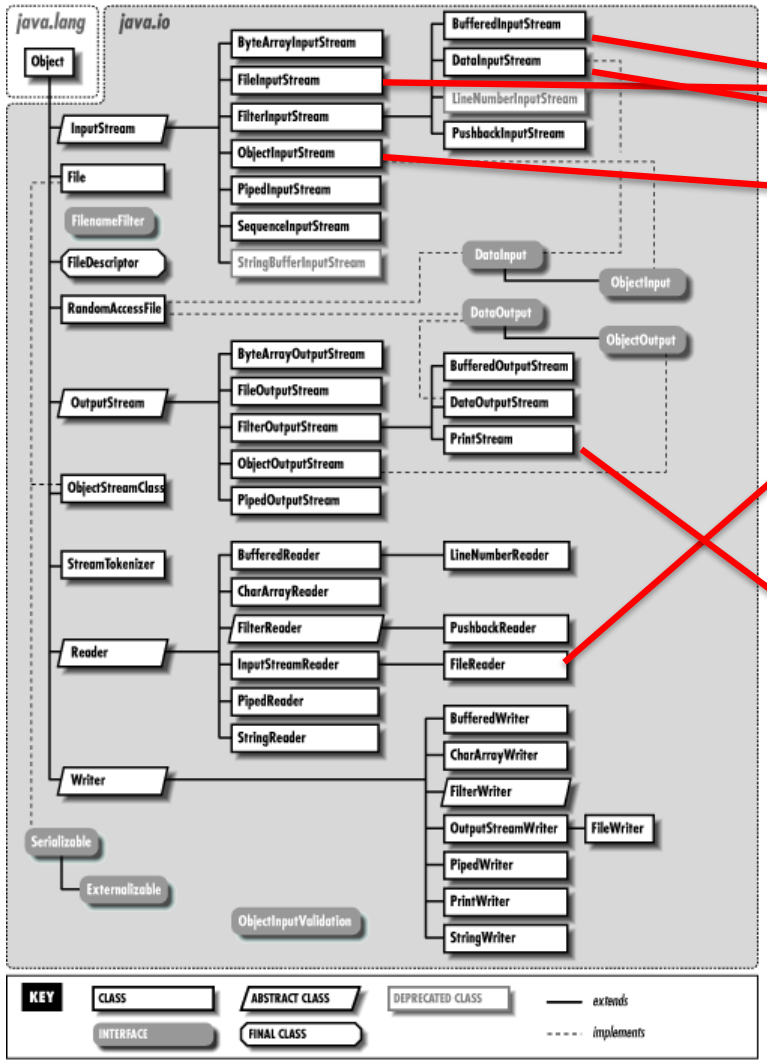
Streams & Threads

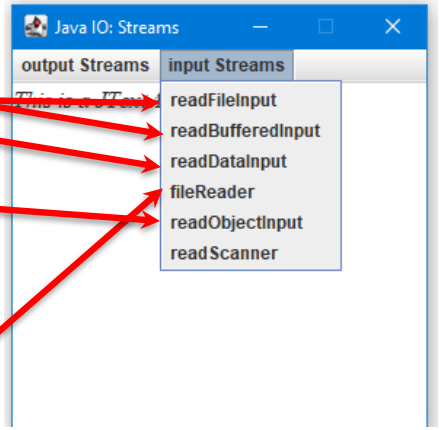
I/O & Streams

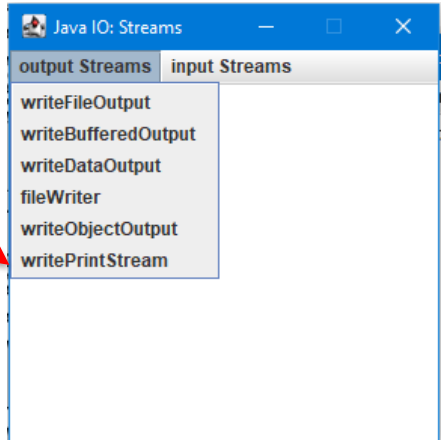
- **I/O:** Input/Output to and from programs. Input can be from keyboard or a file. Output can be to display (screen) or a file.

- **Stream**: An object that either delivers data to its destination (screen, file, etc.) or that takes data from a source (keyboard, file, etc.)
- **Output stream**: A stream that accepts output from a program
  - `System.out` is an output stream. Connects a program to the screen
- **Input stream**: A stream that provides input to a program
  - `System.in` is an input stream. Connects a program to the keyboard

Types of Streams & our Program







`read(), write(), close()`

Simple & Processing Streams

- **FileOutputStream**: An output stream for writing data to a File. FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter.
- **FileInputStream:** Obtains input bytes from a file in a file system. FileInputStream is meant for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

- **FileWriter:** Convenience class for writing character files. FileWriter is meant for writing streams of characters.
- **FileReader:** Convenience class for reading character files. FileReader is meant for reading streams of characters.

- **BufferedOutputStream:** A buffered output stream. By setting up such an output stream, an application can write bytes to the underlying output stream without necessarily causing a call to the underlying system for each byte written.
- **BufferedInputStream:** Adds functionality to another input stream-namely, the ability to buffer the input. When the BufferedInputStream is created, an internal buffer array is created. As bytes from the stream are read or skipped, the internal buffer is refilled as necessary from the contained input stream, many bytes at a time.

- **DataOutputStream:** Lets an application write primitive Java data types to an output stream in a portable way.
- **DataInputStream:** Lets an application read primitive Java data types from an underlying input stream in a machine-independent way.

- **PrintStream:** Adds functionality to another output stream, namely the ability to print representations of various data values conveniently. All characters printed by a PrintStream are converted into bytes. The PrintWriter class should be used in situations that require writing **characters** rather than bytes.
- **Scanner:** A simple text scanner can parse primitive types and strings using regular expressions.

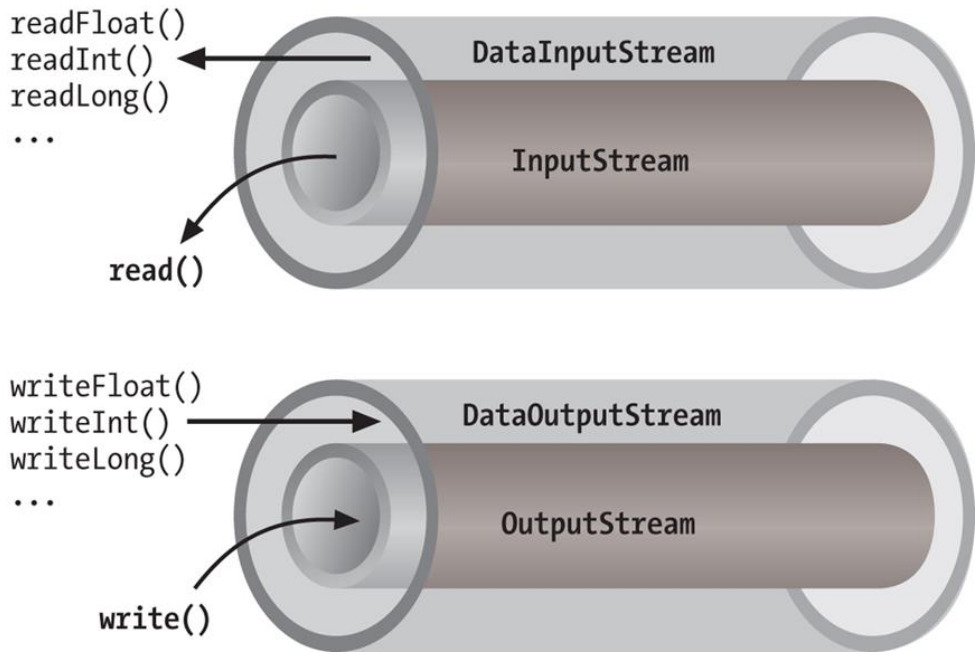
Simple Streams and Processing Streams (Wrappers)

Simple:

```
FileOutputStream fileOutputStream =new FileOutputStream(file);
```

Processing:

```
DataOutputStream dataOutputStream =
    new DataOutputStream(fileOutputStream);
```



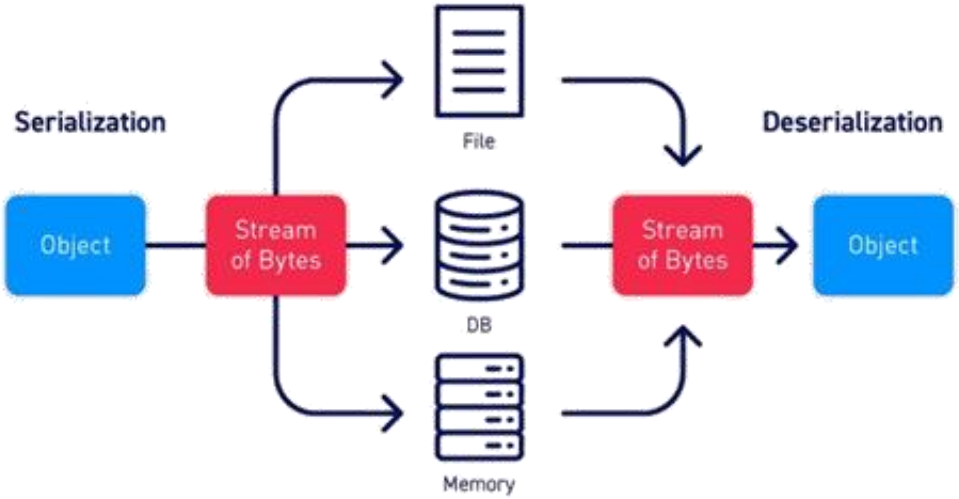
Serialization

- **ObjectOutputStream:** Writes primitive data types and graphs of Java objects to an OutputStream.
- **ObjectInputStream:** Deserializes primitive data and objects previously written using an ObjectOutputStream.

- Only objects that support the java.io.Serializable or java.io.Externalizable interface can be read or written from streams. Implementing the Serializable interface allows object serialization to save and restore the **entire state** of the object and it allows classes to evolve between the time the stream is written and the time it is read. It automatically traverses references between objects, saving and restoring entire graphs.

- The default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields.
- Fields declared as transient or static are ignored by the deserialization process.

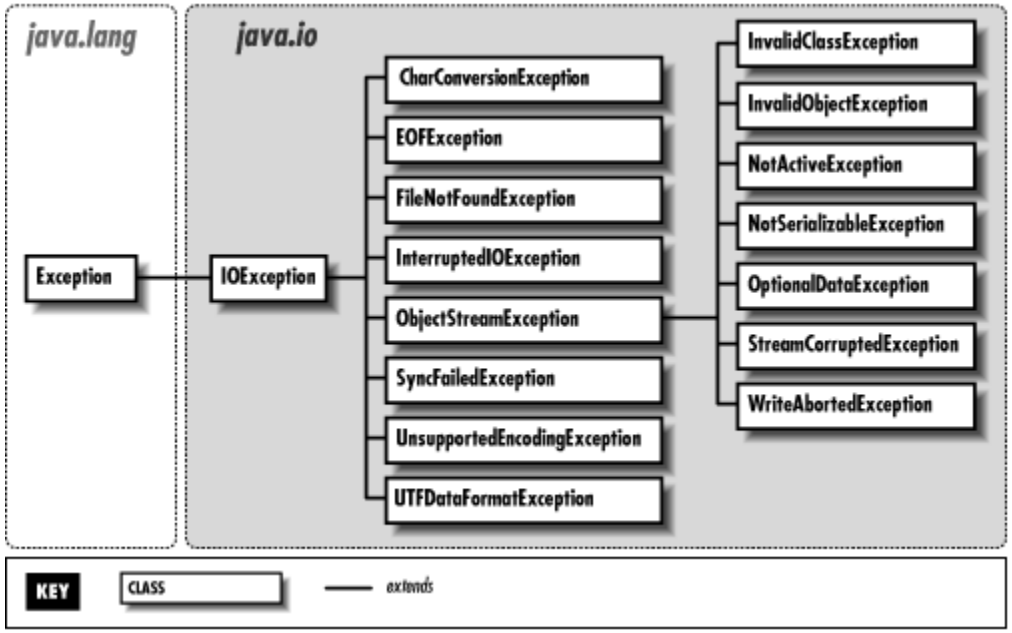
- **Serialization** is the conversion of the state of an object into a byte stream; deserialization does the opposite. Stated differently, serialization is the conversion of a Java object into a static stream (sequence) of bytes which can then be saved to a database or transferred over a network.



Exceptions

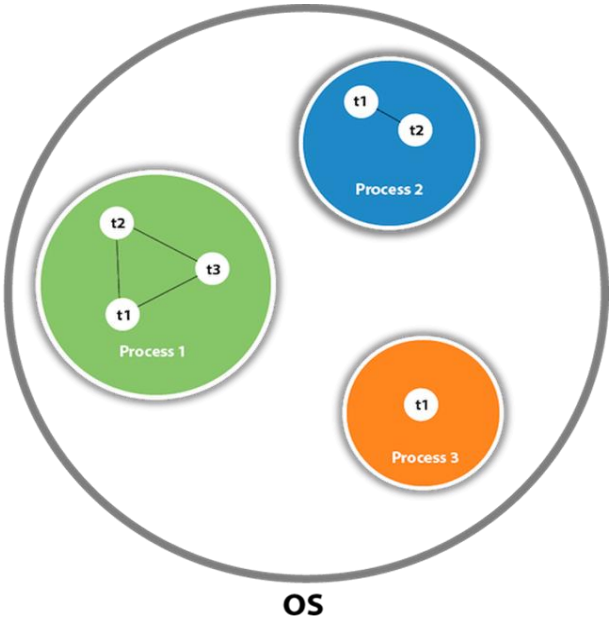
- An **exception** (or exceptional event) is a problem that arises during the execution of a program. When an Exception occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.
- The **finally** statement lets you execute code, after try and catch, regardless of the result.
- Use the **throw** statement to create a custom error (throw an exception).



Threads

- A thread is a lightweight sub-process, the smallest unit of processing.
- Threads allow a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.



Creating a Thread

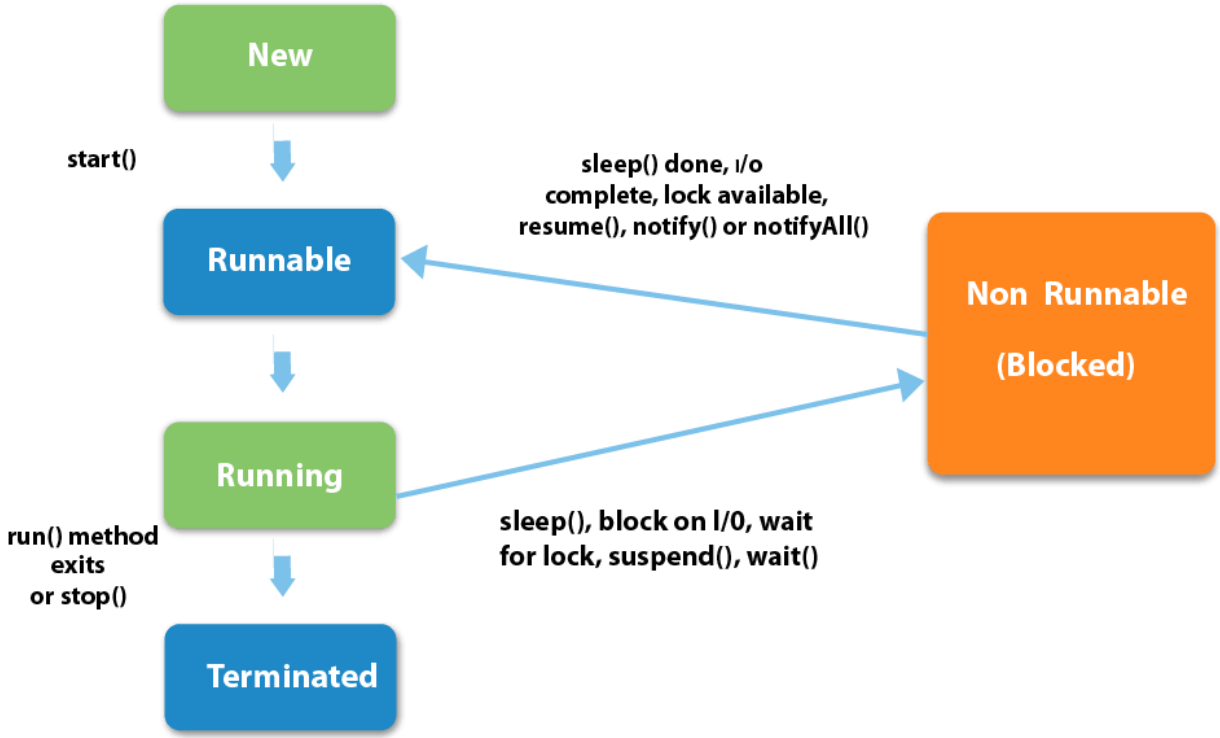
- By **extending** the Thread class and overriding its `run()` method:

```
public class Main extends Thread {
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

- By **implementing** the Runnable interface:

```
public class Main implements Runnable {
    public void run() {
        System.out.println("This code is running in a thread");
    }
}
```

Thread States

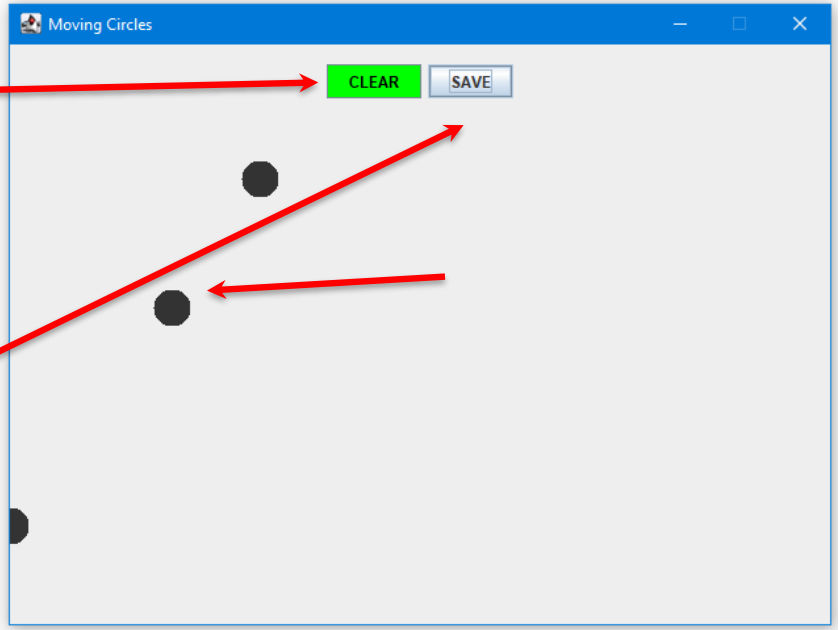


Our Program

Markers | Properties | Servers | Data Source Explorer | Snippets | Console

myThreads [Java Application] C:\Internet and Applications Lab\ECLIPSE\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_15.0.1.v20201027-0507\jre\bin\javaw.exe (13 Μαΐ 2021, 9:22:05 π.μ.)

```
Starting hello thread...
Starting newcircle thread...
Starting thread3...
Starting information thread...
Hello
The timer is working
Hello
Hello
NewCircle
INFORMATION
myCircles:  2

Using Iterator
myCicle{X='165',Y='231',velX='-5', velY='5', RAD='10'}

myCicle{X='48',Y='15',velX='-5', velY='5', RAD='12'}

[myCicle{X='329',Y='322',velX='-5', velY='-5', RAD='10'}
, myCicle{X='13',Y='151',velX='-5', velY='5', RAD='12'}
]
Hello
Hello
BUTTON SAVE  !!!!
Hello
NewCircle
myCircles:  3

Using Iterator
myCicle{X='65',Y='331',velX='-5', velY='5', RAD='10'}

myCicle{X='48',Y='115',velX='-5', velY='5', RAD='12'}

myCicle{X='265',Y='...',velX='-5', velY='5', RAD='14'}

INFORMATION
[myCicle{X='95',Y='301',velX='-5', velY='5', RAD='10'}
, myCicle{X='18',Y='85',velX='5', velY='5', RAD='12'}
]
Hello
Hello
The timer is working
Hello
```



Thread Methods: Synchronization

- The **join()** is a synchronization method that blocks the calling thread (that is, the thread that calls the method) until the thread whose join method is called has completed. Use this method to ensure that a thread has been terminated.

- The **wait()** method causes the current thread to wait indefinitely until another thread either invokes **notify()** for this object or **notifyAll()**.

- **sleep()** is used for time-synchronization, whereas **wait()** is used for multi-thread-synchronization.

Questions

- **Site**: http://ecourses.dbnet.ntua.gr/15373.html
- Email: dkmsgroup@gmail.com
- Slack: https://join.slack.com/t/internetappli-qob4034/signup
  - Register using your email at mail.ntua.gr