Advanced Programming Methods

Lecture 7 - Concurrency in Java (2) and Java Serialization

Content(java.util.concurrent)

1)**Executor Service**

2)**ForkJoinPool**

3)**Blocking Queue**

4)**Concurrent Collections**

5)**Semaphore**

6)**CountDownLatch**

7)**CyclicBarrier**

8)**Lock**

9)**Atomic Variables**

Java.util.concurrency

- A a set of ready-to-use data structures and functionality for writing safe multithreaded applications

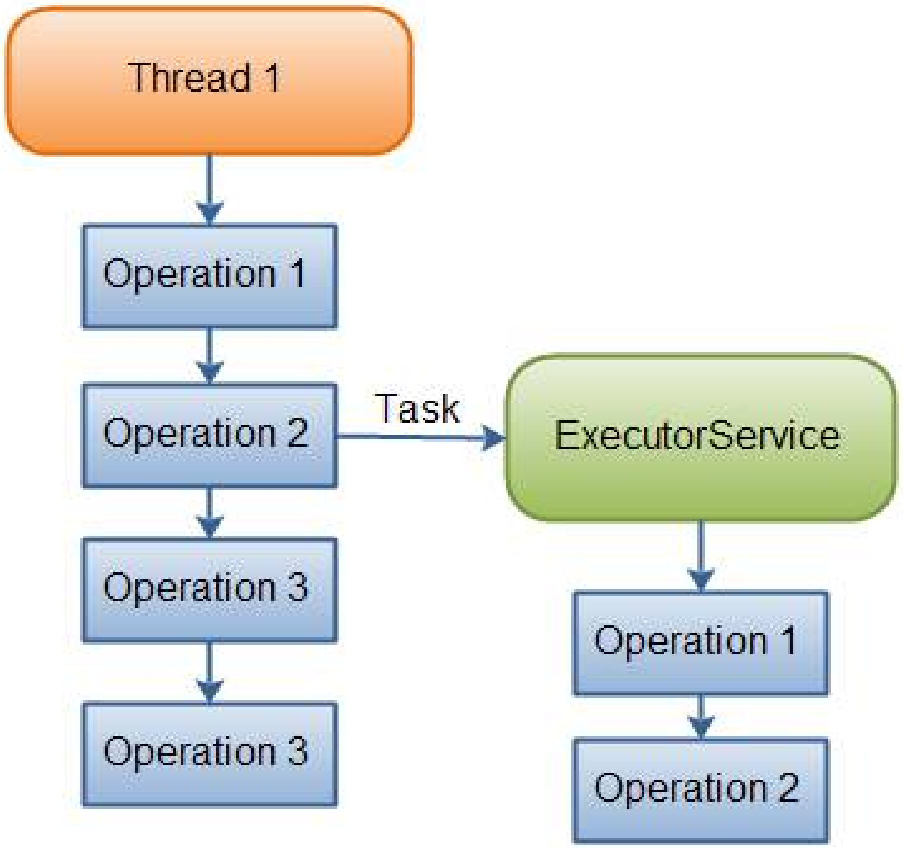
- it is **not always trivial** to write robust code that executes well in a multi-threaded environment

Executor Service

- The java.util.concurrent.ExecutorService interface represents an asynchronous execution mechanism which is capable of executing tasks in the background.
- It is very similar to a thread pool. In fact, the implementation of ExecutorService present in the java.util.concurrent package is a thread pool implementation.

ExecutorService

```
ExecutorService executorService
   =Executors.newFixedThreadPool(10);
//10 threads for executing tasks are created


executorService.execute(new Runnable() {
   public void run() {
      System.out.println("Asynchronous task");
   }
});


executorService.shutdown();
```



Thread 1 delegates a task to an Executor Service for asynchronous execution

Creating executor service

Using Executors factory class:

**ExecutorService executorService1 =
Executors.newSingleThreadExecutor();**

**ExecutorService executorService2 =
Executors.newFixedThreadPool(10);**

**ExecutorService executorService3 =
Executors.newScheduledThreadPool(10);**

ExecutorService usage

There are a few different ways to delegate tasks for execution to an ExecutorService:

- **execute(Runnable)**
- **submit(Runnable)**
- **submit(Callable)**
- **invokeAny(...)**
- **invokeAll(...)**

execute(Runnable)

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();

executorService.execute(new Runnable() {

   public void run() {

       System.out.println("Asynchronous task");

   }

});

executorService.shutdown();
```

- There is no way of obtaining the result of the executed Runnable, if necessary. We have to use a Callable for that

submit(Callable)

```
Future future = executorService.submit(new Callable(){
    public Object call() throws Exception {
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});
System.out.println("future.get() = " + future.get());
```

- The Callable's result can be obtained via the Future object returned

invokeAll()

```
ExecutorService executorService =
    Executors.newSingleThreadExecutor();

Set<Callable<String>> callables = new HashSet<Callable<String>>();

callables.add(new Callable<String>() {

   public String call() throws Exception {

      return "Task 1";

   }

});

......

List<Future<String>> futures = executorService.invokeAll(callables);

for(Future<String> future : futures){

   System.out.println("future.get = " + future.get());

}

executorService.shutdown();
```

```
ExecutorService executor = Executors.newWorkStealingPool();
List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");
executor.invokeAll(callables)
    .stream()
    .map(future -> {
        try {
            return future.get();
        }
        catch (Exception e) {
            throw new IllegalStateException(e);
        }
    })
    .forEach(System.out::println);
```

Callables and Futures

- Callables are functional interfaces just like runnables but instead of being void they return a value.

```
Callable<Integer> task = () -> {
    try {
        TimeUnit.SECONDS.sleep(1);
        return 123;
    }
    catch (InterruptedException e) {
        throw new IllegalStateException("task interrupted", e);
    }
};
```

Callables and Futures

- Callables can be submitted to executor services just like runnables.
- the executor returns a special result of type Future which can be used to retrieve the actual result at a later point in time.
- After submitting the callable to the executor we can check if the future has already been finished execution via isDone()

```
ExecutorService executor = Executors.newFixedThreadPool(1);

Future<Integer> future = executor.submit(task);

System.out.println("future done? " + future.isDone());

Integer result = future.get();

System.out.println("future done? " + future.isDone());

System.out.print("result: " + result);


//future done? false

//future done? true

//result: 123
```

ExecutorService Shutdown

- To terminate the threads inside the ExecutorService you call its shutdown() method. It will not shut down immediately, but it will no longer accept new tasks, and once all threads have finished current tasks, the ExecutorService shuts down

- to shut down the ExecutorService immediately, you can call the shutdownNow() method. This will attempt to stop all executing tasks right away, and skips all submitted but non-processed tasks.

java.util.concurrent.ThreadPoolExecutor

- is an implementation of the ExecutorService interface.
- executes the given task (Callable or Runnable) using one of its internally pooled threads.

- The number of threads in the pool is determined by these variables:
  - corePoolSize
  - maximumPoolSize

ScheduledExecutorService

- It is an interface
- can schedule tasks to run after a delay, or to execute repeatedly with a fixed interval of time in between each execution.
- Tasks are executed asynchronously by a worker thread, and not by the thread handing the task to the ScheduledExecutorService.

ScheduledExecutorService

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(5);

ScheduledFuture scheduledFuture =
    scheduledExecutorService.schedule(new Callable() {
        public Object call() throws Exception {
            System.out.println("Executed!");
            return "Called!";
        }
    }, 5,TimeUnit.SECONDS);
```

- the Callable should be executed after 5 seconds

ScheduledExecutorService Usage

Once you have created a ScheduledExecutorService you use it by calling one of its methods:

- schedule (Callable task, long delay, TimeUnit timeunit)
- schedule (Runnable task, long delay, TimeUnit timeunit)
- scheduleAtFixedRate (Runnable, long initialDelay, long period, TimeUnit timeunit)
- scheduleWithFixedDelay (Runnable, long initialDelay, long period, TimeUnit timeunit)

ForkJoinPool

- is similar to the ExecutorService but with one difference

- implements the work-stealing strategy, i.e. every time a running thread has to wait for some result; the thread removes the current task from the work queue and executes some other task ready to run. This way the current thread is not blocked and can be used to execute other tasks. Once the result for the originally suspended task has been computed the task gets executed again and the join() method returns the result.

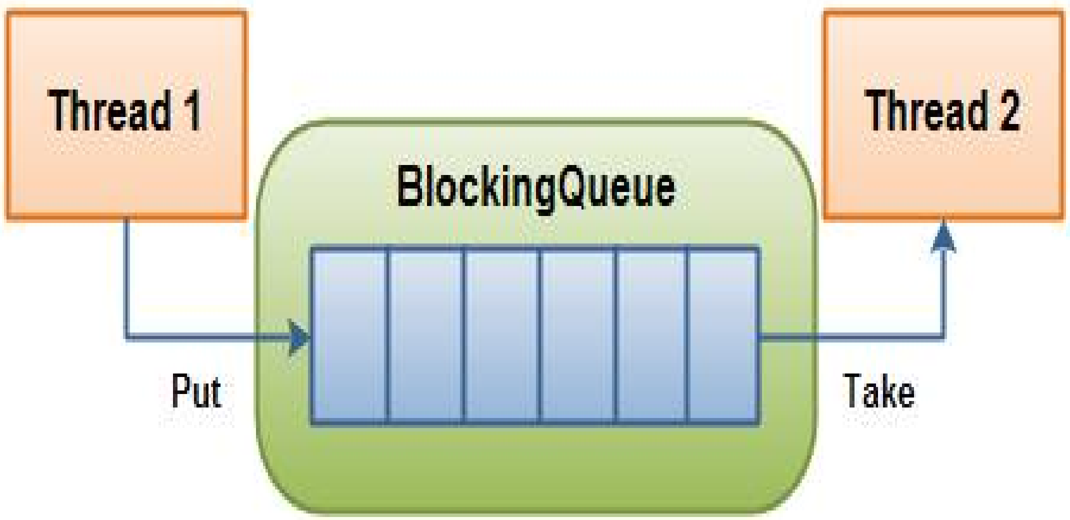
ForkJoinPool

- a call of fork() will start an asynchronous execution of the task,
- a call of join() will wait until the task has finished and retrieve its result.
- makes it easy for tasks to split their work up into smaller tasks(divide and conquer approach) which are then submitted to the ForkJoinPool too.

```java
public class FindMin extends RecursiveTask<Integer> {
private static final long serialVersionUID = 1L;
private int[] numbers;
private int startIndex;
private int endIndex;

public FindMin(int[] numbers, int startIndex, int endIndex) {
this.numbers = numbers;
this.startIndex = startIndex;
this.endIndex = endIndex;
}

@Override
protected Integer compute() {
int sliceLength = (endIndex - startIndex) + 1;
if (sliceLength > 2) {
FindMin lowerFindMin = new FindMin(numbers, startIndex, startIndex + (sliceLength / 2)- 1);
lowerFindMin.fork();
```

```
FindMin upperFindMin = new FindMin(numbers, startIndex + (sliceLength / 2), endIndex);
upperFindMin.fork();
return Math.min(lowerFindMin.join(), upperFindMin.join());
} else {
return Math.min(numbers[startIndex], numbers[endIndex]);
}
}

public static void main(String[] args) {
int[] numbers = new int[100];
Random random = new Random(System.currentTimeMillis());
for (int i = 0; i < numbers.length; i++) {
numbers[i] = random.nextInt(100);
}
ForkJoinPool pool = new ForkJoinPool(Runtime.getRuntime().availableProcessors());
Integer min = pool.invoke(new FindMin(numbers, 0, numbers.length - 1));
System.out.println(min);
}
}
```

Interface BlockingQueue



Interface BlockingQueue

- methods come in four forms, with different ways of handling operations that cannot be satisfied immediately, but may be satisfied at some point in the future:
  - one throws an exception: **add(e), remove(), element()**
  - the second returns a special value (either null or false, depending on the operation): **offer(e), poll(), peek()**
  - the third blocks the current thread indefinitely until the operation can succeed: **put(e), take()**
  - the fourth blocks for only a given maximum time limit before giving up: **offer(e, time, unit), poll(time, unit)**

```
public class BlockingQueueExample {

    public static void main(String[] args) throws Exception {

        BlockingQueue queue = new ArrayBlockingQueue(1024);

        Producer producer = new Producer(queue);
        Consumer consumer = new Consumer(queue);

        new Thread(producer).start();
        new Thread(consumer).start();

        Thread.sleep(4000);
    }
}
```

```java
public class Producer implements Runnable{
    protected BlockingQueue queue = null;
    public Producer(BlockingQueue queue) {
        this.queue = queue;}

    public void run() {
        try {
            queue.put("1");
            Thread.sleep(1000);
            queue.put("2");
            Thread.sleep(1000);
            queue.put("3");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

```java
public class Consumer implements Runnable{
    protected BlockingQueue queue = null;
    public Consumer(BlockingQueue queue) {
        this.queue = queue; }

    public void run() {
        try {
            System.out.println(queue.take());
            System.out.println(queue.take());
            System.out.println(queue.take());
        } catch (InterruptedException e) {
            e.printStackTrace();
        }}}
```

ConcurrentHashMap

- is very similar to the java.util.HashTable class, except that ConcurrentHashMap offers better concurrency than HashTable does.

- does not lock the Map while you are reading from it.

- does not lock the entire Map when writing to it. It only locks the part of the Map that is being written to, internally.

Semaphore

- the java.util.concurrent.Semaphore class is a counting semaphore.
- The counting semaphore is initialized with a given number of "permits".
- For each call to acquire() a permit is taken by the calling thread.
- For each call to release() a permit is returned to the semaphore.
- Thus, at most N threads can pass the acquire() method without any release() calls, where N is the number of permits the semaphore was initialized with.

Semaphore

As semaphore typically has two uses:

- To guard a critical section against entry by more than N threads at a time.
- To send signals between two threads.

```
ExecutorService executor = Executors.newFixedThreadPool(10);
Semaphore semaphore = new Semaphore(5);
Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {System.out.println("Could not acquire semaphore");}
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {semaphore.release();
        }}}

IntStream.range(0, 10).forEach(i -> executor.submit(longRunningTask));
stop(executor);
```

CountDownLatch

- is initialized with a given count.

- This count is decremented by calls to the countDown() method.

- Threads waiting for this count to reach zero can call one of the await() methods. Calling await() blocks the thread until the count reaches zero.

```
CountDownLatch latch = new CountDownLatch(3);

Waiter      waiter      = new Waiter(latch);
Decrementer decrementer = new Decrementer(latch);

new Thread(waiter)     .start();
new Thread(decrementer).start();

Thread.sleep(4000);
```

```
public class Waiter implements Runnable{
    CountDownLatch latch = null;
   public Waiter(CountDownLatch latch) {
       this.latch = latch;}
   public void run() {
       try {
           latch.await();
       } catch (InterruptedException e) {
           e.printStackTrace();
       }
       System.out.println("Waiter Released");
   }
}
```

```
public class Decrementer implements Runnable {
  CountDownLatch latch = null;
   public Decrementer(CountDownLatch latch) {
      this.latch = latch;}
   public void run() {
      try {
         Thread.sleep(1000);
         this.latch.countDown();
         Thread.sleep(1000);
         this.latch.countDown();
         Thread.sleep(1000);
         this.latch.countDown();
      } catch (InterruptedException e) {
         e.printStackTrace();
      }}}
```

Cyclic Barrier

- is a synchronization mechanism that can synchronize threads progressing through some algorithm.
- it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue.
- The threads wait for each other by calling the await() method on the CyclicBarrier.
- Once N threads are waiting at the CyclicBarrier, all threads are released and can continue running.

```
Runnable barrier1Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 1 executed ");}};
Runnable barrier2Action = new Runnable() {
    public void run() {
        System.out.println("BarrierAction 2 executed ");}};

CyclicBarrier barrier1 = new CyclicBarrier(2, barrier1Action);
CyclicBarrier barrier2 = new CyclicBarrier(2, barrier2Action);
CyclicBarrierRunnable barrierRunnable1 =
        new CyclicBarrierRunnable(barrier1, barrier2);
CyclicBarrierRunnable barrierRunnable2 =
        new CyclicBarrierRunnable(barrier1, barrier2);
new Thread(barrierRunnable1).start();
new Thread(barrierRunnable2).start();
```

```
public class CyclicBarrierRunnable implements Runnable{

    CyclicBarrier barrier1 = null;
    CyclicBarrier barrier2 = null;

    public CyclicBarrierRunnable(
            CyclicBarrier barrier1,
            CyclicBarrier barrier2) {

        this.barrier1 = barrier1;
        this.barrier2 = barrier2;
    }
```

```java
public void run() {
    try {
        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() +
                        " waiting at barrier 1");
        this.barrier1.await();
        Thread.sleep(1000);
        System.out.println(Thread.currentThread().getName() +
                        " waiting at barrier 2");
        this.barrier2.await();
        System.out.println(Thread.currentThread().getName() +  " done!");
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }}}
```

Lock

- is a thread synchronization mechanism just like synchronized blocks. A Lock is, however, more flexible and more sophisticated than a synchronized block.

```
Lock lock = new ReentrantLock();
lock.lock();
//critical section
lock.unlock();
```

ReadWriteLock

- allows multiple threads to read a certain resource, but only one to write it, at a time.
- Read Lock: If no threads have locked the ReadWriteLock for writing, and no thread have requested a write lock. Thus, multiple threads can lock the lock for reading.
- Write Lock: If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing.

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();


readWriteLock.readLock().lock();
    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.
readWriteLock.readLock().unlock();


readWriteLock.writeLock().lock();
    // only one writer can enter this section,
    // and only if no threads are currently reading.
readWriteLock.writeLock().unlock();
```

Atomic Variables

- the atomic classes make heavy use of compare-and-swap (CAS), an atomic instruction directly supported by most modern CPUs.
- Those instructions usually are much faster than synchronizing via locks.
- The advice is to prefer atomic classes over locks in case you just have to change a single mutable variable concurrently.
- Many atomic classes: AtomicBoolean, AtomicInteger, AtomicReference, AtomicIntegerArray, etc

AtomicBoolean

- provides a boolean variable which can be read and written atomically, and which also contains advanced atomic operations like compareAndSet()
- Example:

```
AtomicBoolean atomicBoolean = new AtomicBoolean(true);

boolean expectedValue = true;

boolean newValue      = false;

boolean wasNewValueSet =
    atomicBoolean.compareAndSet(expectedValue, newValue);
```

AtomicInteger

```
AtomicInteger atomicInt = new AtomicInteger(0);
ExecutorService executor = Executors.newFixedThreadPool(2);
IntStream.range(0, 1000)
    .forEach(i -> {
        Runnable task = () ->
            atomicInt.updateAndGet(n -> n + 2); //thread-safe without synchronization
        executor.submit(task);
    });

stop(executor);
System.out.println(atomicInt.get());    // => 2000
```

Java Serialization

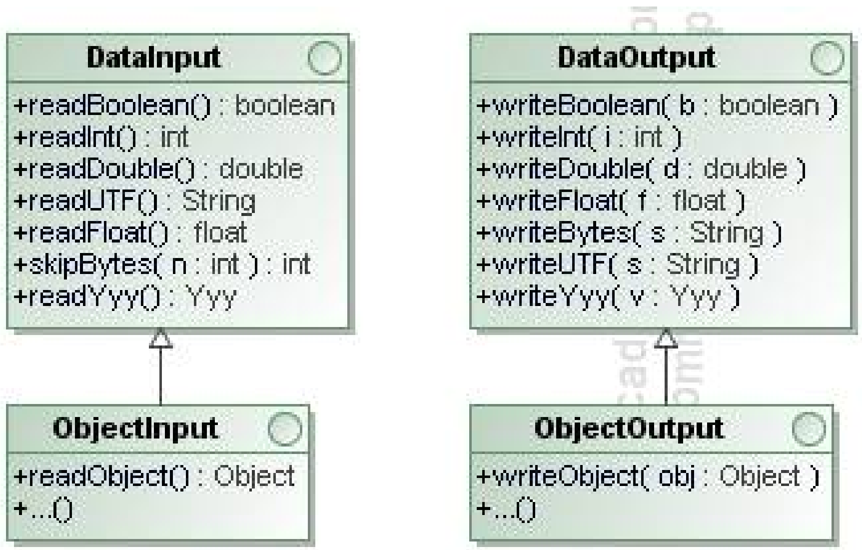
Java Serialization

- serialization allows us to convert the state of an object into a byte stream, which then can be saved into a file on the local disk or sent over the network to any other machine.

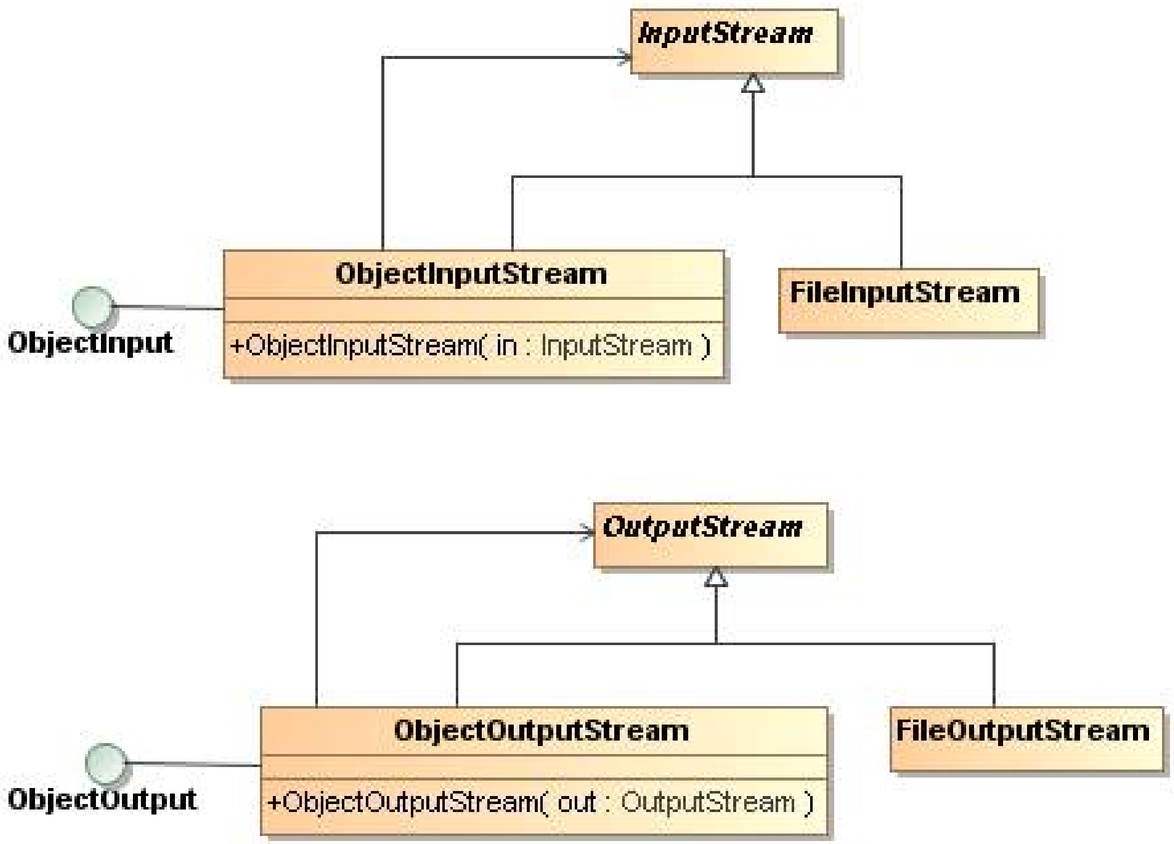
- deserialization allows us to reverse the process, which means reconverting the serialized byte stream to an object again.

Java Objects Serialization

- The process of writing/reading objects from/to a file/external support.
- An object is persistent (serializable) if it can be written into a file/external support and can be read from a file/external support



Objects Serialization



Objects Serialization

```
void serializareObj(String numefis){
        ObjectOutputStream out=null;
        try{
            out=new ObjectOutputStream(new FileOutputStream(numefis));
            out.writeObject(23);
            out.writeObject("Vasilescu Ana");
            out.writeObject(23.45f);
        } catch (IOException e) {
            System.err.println("Eroare "+e);
        } finally {
            if (out!=null)
                try {
                    out.close();
                } catch (IOException e) {
                    System.err.println("Eroare "+e);
                }
        }
    }
```

Objects Serialization

```
void deserializareObj(String numefis){
  ObjectInputStream in=null;
  try{
     in=new ObjectInputStream(new FileInputStream(numefis));
     Integer intreg=(Integer)in.readObject();
     String text=(String)in.readObject();
     Float nr=(Float)in.readObject();
     System.out.println("Intreg: "+intreg+" String: "+text+" Float: "+nr);
  } catch (IOException e) {System.err.println("Eroare "+e);}
   catch (ClassNotFoundException e) {
     System.err.println("Eroare deserializare "+e);
   }finally {
       if (in!=null){
          try {
              in.close();
          } catch (IOException e) {System.err.println("Eroare "+e);}


        }
    }
}
```

Serializable Objects

- The classes whose objects are serializable must be declared to implement the interface `Serializable` (package `java.io`).
- Interface Serializable does not contain any method.

```
class Student implements Comparable<Student>, Serializable{
 //...
}
class Test{
  public static void main(String[] args){
     ObjectOutputStream out=
      //... initialization
      Student stud=new Student("Popescu Ioan", 7.9);
      out.writeObject(stud);
      //...
  }
}
```

- The state of `stud` (the values of its fields) is saved into the file.

Serializable objects

- All the reachable objects (the objects that can be reach using the references) are saved into the file only once.

```
class CircularList implements Serializable{
  private class Node implements Serializable{
      Node urm;
      //...
  }
   private Node head; //last node of the list refers to the head of the list
   //...
}
```

- The objects which are referred by a serializable object must be also serializable.

Obs:

Static attributes of a serializable class are not saved into the file/external support.

Example serializable objects

```
void printSerializabil(List<Student> studs, String numefis){
        ObjectOutputStream out=null;
        try{
          out=new ObjectOutputStream(new FileOutputStream(numefis));
            out.writeObject(studs);
        } catch (IOException e) {
            System.err.println("Eroare serializare "+e );
        } finally {
            if (out!=null)
                try {
                    out.close();
                } catch (IOException e) {
                    System.err.println("Eroare "+e);
                }
        }
    }
```

Example serializable objects

```
@SuppressWarnings("unchecked")
List<Student> citesteSerializabil(String numefis){
        List<Student> rez=null;
        ObjectInputStream in=null;
        try{
            in=new ObjectInputStream(new FileInputStream(numefis));
            rez=(List<Student>)in.readObject();
        } catch (IOException e) {
            System.err.println("Eroare deserializare"+e);
        } catch (ClassNotFoundException e) {
            System.err.println("Eroare deserializare "+e);
        }finally{
          if (in!=null)
                try {
             in.close();
                }catch (IOException e) {System.err.println("Eroare "+e); }
       }
        return rez;
}
```

Objects Serialization

- Method `in.readObject():Object`
  1. Read the object from the stream
  2. Identify the object type
  3. Initialize the non-static members of the object byte by byte (without a constructor call) and then return the new created object
- Method `out.writeObject(Object)`
  - Save the non-static members and the information required by JVM to rebuild the object
  - an object (from a given reference) is saved only once on a stream:

```
ObjectOutputStream out=...
out.writeObject(new Produs("A"));
Produs produs2=new Produs("B");
out.writeObject(produs2);
produs2.setNume("BB");
out.writeObject(produs2);
//...
out.close();
```

```
ObjectInputStream in=...
Produs p1=(Produs)in.readObject();
Produs p2=(Produs)in.readObject();
Produs p3=(Produs)in.readObject();
//...
```

Objects Serialization - serialVersionUID

```
public class Student implements Serializable{
  private String nume;
  private double media;
  //...
}
```

Scenario:

1. The objects of class Student are serialized.
2. The class Student is changed (add/remove fields/methods).
3. We want to de-serialize the saved objects.

```
public class Student implements Serializable{
  [any modif access] static final long serialVersionUID = 1L;
   private String nume;
  private double media;
   private int grupa;
  //...
}
```

New added fields are initialized with the default values corresponding to their types.

Objects Serialization - transient

- There are situations when we do not want to save the values of some fields (e.g. passwords, file descriptors, etc.)
- Those fields are declared using the keyword `transient:`

```
public class Student implements Serializable{
  private String nume;
  private double media;
  private transient String parola;
  //...
}
```

At reading, the transient fields are initialized with the default values corresponding to their types.

Serializable data structures

```
public class Stack implements Serializable{
   private class Node implements Serializable{
   //...
   }
    private Node top;
   //...
 }
//...
Stack s=new Stack();
 s.push("ana");
 s.push(new Produs("Paine", 2.3));
                  //class Produs must be serializable
//...
 ObjectOuputStream out=...
  out.writeObject(s);
```