Object pool design pattern

Ardelean Octavian

30432

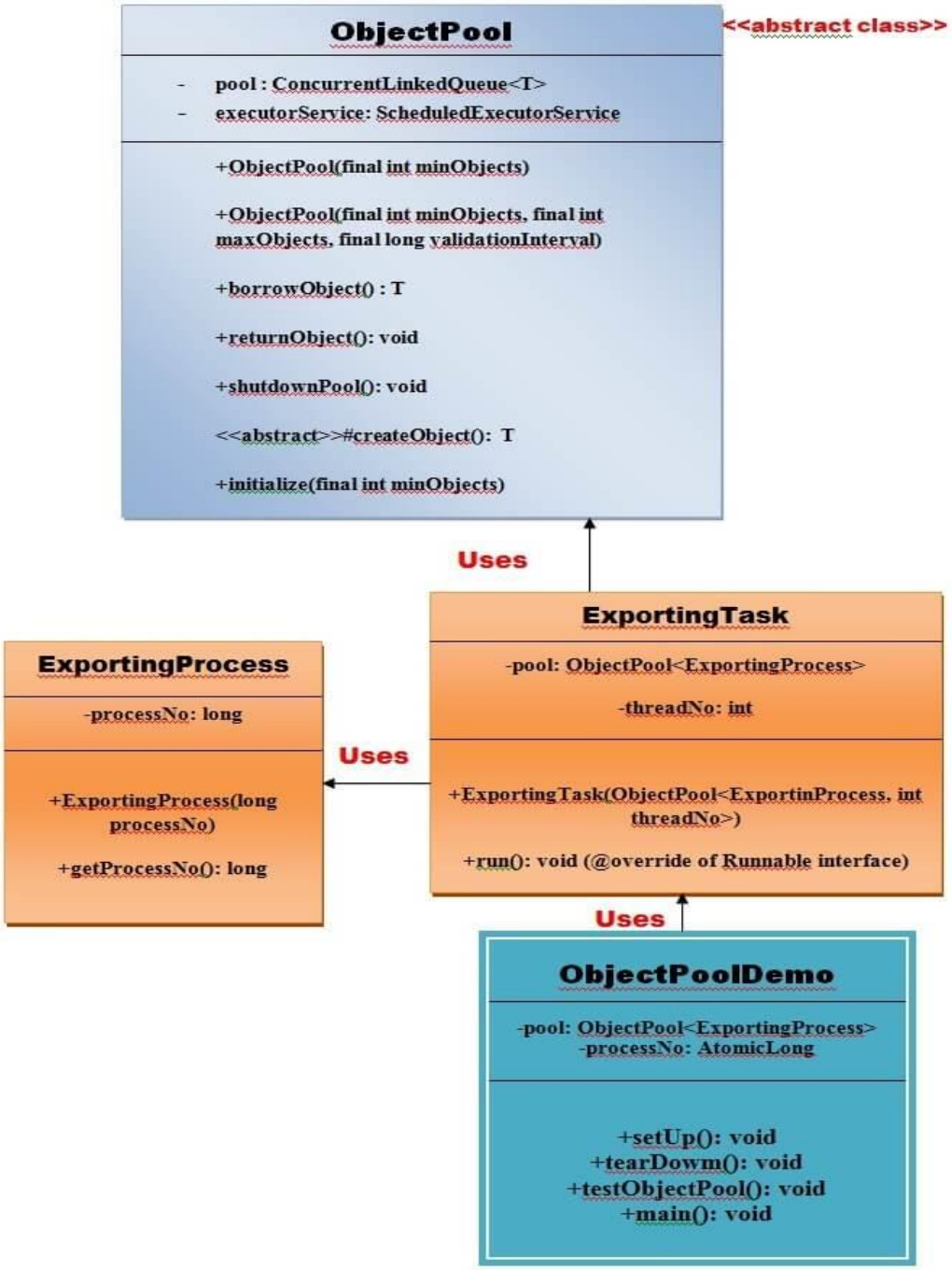
Introduction

- creational design pattern
- uses a set of initialized objects kept ready to use – a "pool" – rather than allocating and destroying them on demand
- reuses the objects that are expensive to create
- objects in the pool have a lifecycle: creation, validation and destroy

When to use

- object pooling is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low.
- Also, when we know that we have a limited number of objects that will be in memory at the same time.

UML DIAGRAM


ObjectPool
<<abstract class>>
- pool : ConcurrentLinkedQueue<T>
- executorService: ScheduledExecutorService
+ObjectPool(final int minObjects)
+ObjectPool(final int minObjects, final int maxObjects, final long validationInterval)
+borrowObject() : T
+returnObject(): void
+shutdownPool(): void
<<abstract>>#createObject(): T
+initialize(final int minObjects)
Uses
ExportingTask
-pool: ObjectPool<ExportingProcess>
-threadNo: int
+ExportingTask(ObjectPool<ExportinProcess, int threadNo>)
+run(): void (@override of Runnable interface)
Uses
ExportingProcess
-processNo: long
+ExportingProcess(long processNo)
+getProcessNo(): long
Uses
ObjectPoolDemo
-pool: ObjectPool<ExportingProcess>
-processNo: AtomicLong
+setUp(): void
+tearDowm(): void
+testObjectPool(): void
+main(): void

EXAMPLE

```
public abstract class ObjectPool<T> {
/*pool implementation based on ConcurrentLinkedQueue from the java.util.concurrent package.
  ConcurrentLinkedQueue is a thread-safe queue based on linked nodes.  */
   private ConcurrentLinkedQueue<T> pool;
    private ScheduledExecutorService executorService;
    public ObjectPool(final int minObjects)
    {initialize(minObjects);}
    public ObjectPool(final int minObjects, final int maxObjects, final long validationInterval) {
        initialize(minObjects);
        executorService = Executors.newSingleThreadScheduledExecutor();
        executorService.scheduleWithFixedDelay(new Runnable()
        {
            @Override
            public void run() {
                int size = pool.size();

                if (size < minObjects) {
                    int sizeToBeAdded = minObjects + size;
                    for (int i = 0; i < sizeToBeAdded; i++) {
                        pool.add(createObject());
                    }
                } else if (size > maxObjects) {
                    int sizeToBeRemoved = size - maxObjects;
                    for (int i = 0; i < sizeToBeRemoved; i++) {
                        pool.poll();
                    }
                }
            }
        }, validationInterval, validationInterval, TimeUnit.SECONDS);
    }
    public T borrowObject() {
        T object;
        if ((object = pool.poll()) == null)  {object = createObject();}
        return object;
    }
    public void returnObject(T object) {
    if(object == null) {return;}
        this.pool.offer(object);
    }
      public void shutdown(){
        if (executorService != null){executorService.shutdown();}
    }
    protected abstract T createObject();
    private void initialize(final int minObjects)  {
        pool = new ConcurrentLinkedQueue<T>();
        for (int i = 0; i < minObjects; i++) {
            pool.add(createObject()); }
            }  }
```

EXAMPLE CONT.

```
public class ExportingProcess {
 private long processNo;
    public ExportingProcess(long processNo)  {
        this.processNo = processNo;
      System.out.println("Object with process no. " + processNo + " was created");
     }

    public long getProcessNo() {
       return processNo;
    }
}
public class ExportingTask implements Runnable {
       private ObjectPool<ExportingProcess> pool;
       private int threadNo;
       public ExportingTask(ObjectPool<ExportingProcess> pool, int threadNo){
           this.pool = pool;
           this.threadNo = threadNo;
       }
       public void run() {
           // get an object from the pool
           ExportingProcess exportingProcess = pool.borrowObject();
           System.out.println("Thread " + threadNo + ": Object with process no. "
                   + exportingProcess.getProcessNo() + " was borrowed");
           //you can  do something here in future
           // .........
           // return ExportingProcess instance back to the pool
           pool.returnObject(exportingProcess);
           System.out.println("Thread " + threadNo +": Object with process no. "
                   + exportingProcess.getProcessNo() + " was returned");
       }
   }// End of the ExportingTask class.
```

```
public class ObjectPoolDemo{
    private ObjectPool<ExportingProcess> pool;
    private AtomicLong processNo=new AtomicLong(0);
    public void setUp() {
    pool = new ObjectPool<ExportingProcess>(4, 10, 5)
      {
          protected ExportingProcess createObject()
          {
              // create a test object which takes some time for creation
              return new ExportingProcess( processNo.incrementAndGet());
          }
      };
}
public void tearDown() {pool.shutdown();}
public void testObjectPool() {
    ExecutorService executor = Executors.newFixedThreadPool(8);
    executor.execute(new ExportingTask(pool, 1));
    executor.execute(new ExportingTask(pool, 2));
    executor.execute(new ExportingTask(pool, 3));
    executor.execute(new ExportingTask(pool, 4));
    executor.execute(new ExportingTask(pool, 5));
    executor.execute(new ExportingTask(pool, 6));
    executor.execute(new ExportingTask(pool, 7));
    executor.execute(new ExportingTask(pool, 8));
    executor.shutdown();
    try {
        executor.awaitTermination(30, TimeUnit.SECONDS);
        } catch (InterruptedException e)

          {  e.printStackTrace();}
}
public static void main(String args[])  {
    ObjectPoolDemo op=new ObjectPoolDemo();
    op.setUp();
    op.tearDown();
    op.testObjectPool();
}
```

Output:

```
Object with process no. 1 was created
Object with process no. 2 was created
Object with process no. 3 was created
Object with process no. 4 was created
Thread 1: Object with process no. 1 was borrowed
Thread 1: Object with process no. 1 was returned
Thread 2: Object with process no. 2 was borrowed
Thread 2: Object with process no. 2 was returned
Thread 3: Object with process no. 3 was borrowed
Thread 3: Object with process no. 3 was returned
Thread 4: Object with process no. 4 was borrowed
Thread 4: Object with process no. 4 was returned
Thread 5: Object with process no. 1 was borrowed
Thread 5: Object with process no. 1 was returned
Thread 6: Object with process no. 2 was borrowed
Thread 6: Object with process no. 2 was returned
Thread 7: Object with process no. 3 was borrowed
Thread 7: Object with process no. 3 was returned
Thread 8: Object with process no. 4 was borrowed
Thread 8: Object with process no. 4 was returned
```

Advantages

- can offer a significant performance boost
- it manages the connections and provides a way to reuse and share them.
- object pool pattern is used when the rate of initializing a instance of the class is high.

Disadvantages

- If the pool is used by multiple threads, it may need the means to prevent parallel threads from grabbing and trying to reuse the same object in parallel.
- Inadequate resetting of objects may also cause an information leak. If an object contains confidential data (e.g. a user's credit card numbers) that isn't cleared before the object is passed to a new client, a malicious or buggy client may disclose the data to an unauthorized party.