**Object Pool**

* **Introduction:**
  + **What is Object Pool?**

In our system if cost of creating an instance of a class is high and we need a large number of objects for this class for short duration, then we can use an object pool.

* + Here we either pre-create objects of the class or collect unused instances in an in memory cache. When code needs an object of this class we provide it from the cache.
  + One of the most complicated patterns to implement efficiently.
* **UML:**

**A screenshot of a cell phone

Description automatically generated**

* **Implementation Steps:**
  + We start by creating class for object pool.
    - A thread-safe caching of objects should be done in pool.
    - Methods to acquire and release objects should be provided and pool should reset cached object before giving them out.
  + The reusable object must provide methods to reset its state upon “release” by code.
  + We have to decide whether to create new pooled objects when pool is empty or to wait until an object becomes available. Choice is influenced by whether the object is tied to a fixed number of external resources.
* **Example UML:**

**A screenshot of a cell phone

Description automatically generated**

* **Implementation & Design Consideration:**

**Implementation Consideration:**

* + Resetting object state should not be costly operation otherwise you may end up losing your performance savings.
  + Pre-caching objects; meaning creating objects in advance can be helpful as it won’t slow down the code using these objects. However it may add-up to start up time and memory consumption.
  + Object pools synchronization should consider the reset time needed and avoid resetting in synchronized context if possible.

**Design Consideration:**

* + Object pool can be parameterized to cache and return multiple objects and the acquire method can provide selection criteria.
  + Pooling objects is only beneficial if they involve costly initialization because of initialization of external resource like connection or a thread. Don’t pool objects JUST to save memory, unless you are running into out of memory issues.
  + Do not pool long lived objects or only to save frequent call to new. Pooling may actually negatively impact performance in such cases.
* **Example:**
  + Using object pool for saving the memory allocation & GC cost is almost deprecated now. JVMs and hardware are now more efficient and have access to more memory now.
  + However it is still a very common pattern when we are interacting with external resources like threads, connections.
  + java.util.concurrent.ThreadPoolExecutor is an example of object pool pattern which pools threads. Even though we can directly use this class, you’ll often use it via ExecutorService interface using method like Executor.newCachedThreadPool();

ExecutorService service = Executor.newCachedThreadPool();

service.submit(()-> System.out.println(Thread.currentThread().getName()));

service.shutdown();

* + Apache commons dbcp library is used for database connection pooling. Class org.apache.commons.dbbcp.BasicDataSource in dbcp package is an example of object pool pattern which pools database connections. This pool is commonly created and exposed via JNDI or as a Spring bean in applications.

**A screenshot of a cell phone

Description automatically generated**

* **Comparison with Prototype:**

|  |  |
| --- | --- |
| **Object Pool** | **Prototype** |
| We have a cached objects that frequently live throughout programs entire run. | Prototype creates object when needed and no caching is done. |
| Code using objects from object pool has to return the objects explicitly to pool. Depending on implementation, failing to return to pool may lead to memory or resource leak. | Once an object is cloned no special treatment is needed by client code and object can be used like any regular object. |

* **Pitfalls:**
  + Successful implementation depends on correct use by the client code. Releasing objects back to pool can be vital or correct working.
  + The reusable object needs to take care of resetting its state in efficient way. Some objects may not be suitable for pooling due to this requirement.
  + Difficult to use in refactoring legacy code as the client code & reusable object both need to be aware of object pool.
  + You have to decide what happens when the pool is empty and there is a demand for an object. You can either wait for an object to become free or create a new object. Both options have issues. Waiting can have severe negative impact on performance.
  + If you create new objects when code asks for an object and none are available then you have to do additional work to maintain or trim the pool size or else you will end up with very large pool.
* **Quiz:**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**

**A screenshot of a social media post

Description automatically generated**