Object Pool Design Pattern

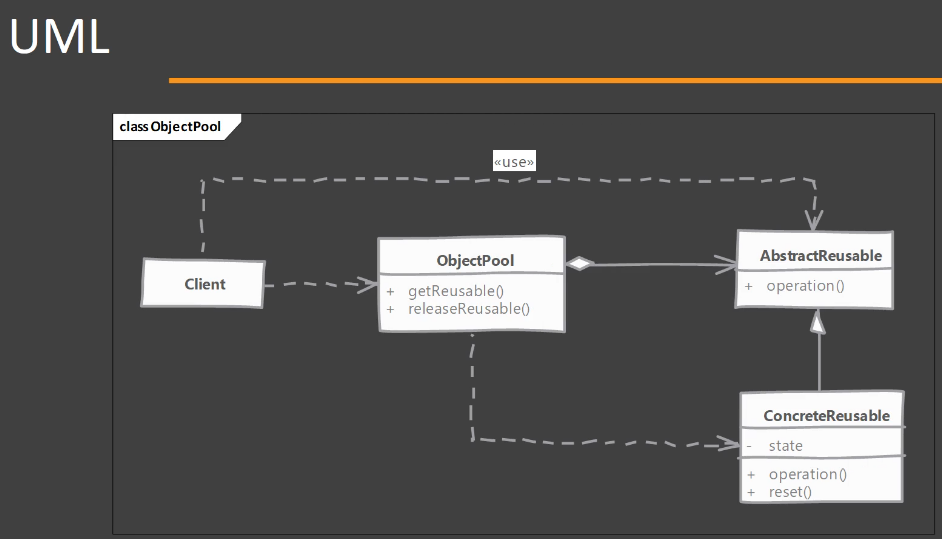
1. Object Pool – Introduction:

Now let's learn about another Creational design pattern called as Object Pool. Now if you have worked with Connection Pool in your JDBC program, or if you have worked with a thread pool, then you have already used an object pool. However, in this lecture we're going to see how we can create our own object pool, which can cache any object that we want. But the important question before we can do that is when you should use an object pool. So let's say you have a class in your application and creating object of that class has a significant cost attached with it. Now, it could be a performance cost or it could be because that object is consuming a limited external resource like threads or for example, a socket on your operating system. So whenever you have such a situation and you still need large number of such objects for very short duration, then you can use an object pool. Now I'm aware I have put up a mouthful of criterion about when you can use an object pool because there are very few scenarios where an object pool can improve performance. As you will see in some time, you can actually hurt your application's performance if you don't implement this design pattern carefully. Now, there are a couple of ways that we can add objects in the object pool. We can either pre-create objects when we are instantiating our object pool, or we can keep on collecting unused instances from our code and cache them in memory so that whenever somebody asks for an object of that particular class, we can return that object from our in-memory cache. Now, this is one of the most complicated patterns to implement efficiently and without defects. And this is true especially because you have to now deal with multithreading issues.

A screenshot of a black and white text

Description automatically generated

So let's look at the UML diagram of this design pattern. Now it's quite a simple UML diagram. First of all, we have an AbstractReusable, so this is the abstract class or an interface which represents our object and client is going to use operations that are defined in this interface or an abstract class. Next, we have an actual concrete class(i.e., ConcreteReusable) which is going to implement these operations(i.e., defined in AbstractReusable) and this class is also going to have its own state. Next, we have the actual object pool(i.e., ObjectPool). Now the operations on the ObjectPool are pretty simple. It needs to provide a method where client can call that method to get an object from the pool, and it should also provide a method which client can call in order to release an unused object. Lastly, we have the Client, and this is going to interact with our ObjectPool in order to get the objects from it and return unused objects to it. So this is the UML diagram for an Object pool design pattern.

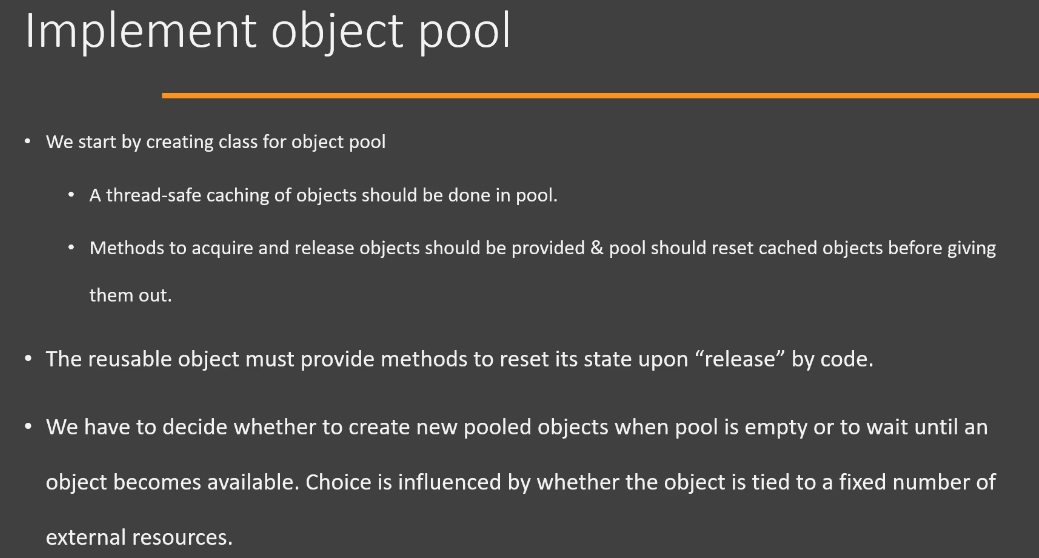


A screenshot of a diagram

Description automatically generated

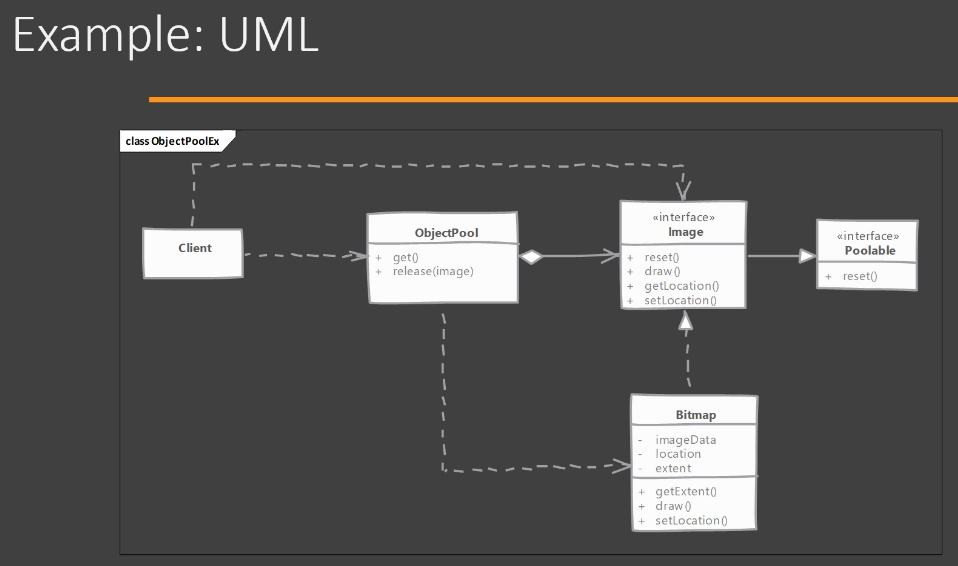
1. Object Pool – Implementation Steps:

Now let's look at some of the steps that we need to follow in order to implement Object Pool design pattern. So we start by creating our class for Object Pool, but it must satisfy a couple of requirements. First requirement is that it should provide a thread-safe caching of objects in memory. Next, it should provide methods that client can call in order to get object from the pool. And another method which client can call in order to return an unused object. Now, one thing that this class must take care is that it should reset the object before giving it to the client. Now that means our reusable object or the class whose objects we are going to pool, it must provide that reset method. Next, there is one choice that you have to make while implementing your Object Pool and that is about what happens when your pool runs out of objects. So you have two choices. You can either create new object if your pool is empty or you can keep on waiting until some object becomes available. Now, this choice is heavily dependent on your implementation. So if your class object(class whose object we are going to pool) is tied with an external resource which is limited in nature then of course you have to wait OR if you have access to unlimited resources or if your object doesn't depend on any external resource then you can keep on creating new objects and then caching them when they are returned to your pool.



1. Object Pool – Hands-on Example UML:

Now let's look at the UML diagram of the example that we're going to implement in Java code. So we start off from the far-right side. You will see we have a interface called as Poolable. So this interface is going to be what our object pool is going to use. So we can have multiple classes that can implement this Poolable interface and they would be eligible to be pooled in our Object Pool implementation. Now this interface(i.e., Poolable) simply defines a single method called as reset() and the purpose of this method, of course, is to reset the internal state of the pooled object. Next, we are going to have an interface called as an Image, and this is going to represent an image that we have loaded from the disk and we want to render it on multiple times. And then we have a concrete class called as Bitmap, which is an implementation of Image, and it has its own state. And the state is going to be the 'imageData'(which is going to be what is loaded from the disk), 'location' and 'extent'. So these are the properties/fields or part of its state that need to be reset before it can be reused. So we can reuse the 'imageData' because that's why we are going to cache this Bitmap, because every time an Image instance is created, we don't want to read from the disk. So that's why we are going to pool this particular Bitmap. So we're going to keep the 'imageData' property as it is and we're going to reset the 'location' and 'extent' properties/fields. Then we're going to have an ObjectPool class. Now, again, this is going to expose two important methods. The first one is the get method that is going to return the pooled Image object, and then it is going to provide another method called as release(), which is going to take an instance of this particular Bitmap and it's going to add that back into the pool. So this is the implementation that we're going to implement in Java.



1. Object Pool – Implementation:

Extract object\_pool\_end\_v2.zip under Design\_patterns\_Java/Creational\_design\_pattern/7\_Object\_Pool\_design\_pattern.

Let's see how we can implement the Object Pool design pattern. So I have a few classes here that I have already implemented. These are exactly the same as we discussed in the UML diagram. I will quickly go over them. We have an Image interface here and this is going to represent an image that can be rendered in our application either on a canvas or directly on the screen. We are not actually going to do the rendering here. Once again, this is a hypothetical application, so let's look at the methods of this interface(i.e., Image). So it has methods that allow someone to set and get the location of this image and an operation that actually draws this image on the location that this Image object will have. Then we have a Bitmap class here, which is an implementation of the Image interface. And this class represents a bitmap file that is stored on our disk(Note: Suppose we will be creating object of Bitmap class corresponding to that particular file only). Now we can see that in the constructor of this class(i.e., Bitmap) we are expecting a string which represents a file name on the disk. So hypothetically, what is going to happen here is that if somebody creates an object of this class, we are going to take this filename and we're going to go to the disk, read that file, allocate a new byte array, and store the file content in that array. We are not actually doing that here, but you get the idea that this is what this class is actually doing, hypothetically. And we want to pool objects of this class(i.e., Bitmap) because every time we create a Bitmap object, it's going to trigger a file I/O, which, you know, might be time consuming and it's going to do a memory allocation and then it's going to store all the file contents in that array. So what we're going to do is we're going to pool objects of this class(i.e., Bitmap) so that we will have some objects in reserve. So if somebody needs an object of this Bitmap class, we can give it an already created object so that it won't trigger/perform the file I/O operation. So that's what we are going to implement in this example.

So to begin with, we are going to code or implement our Object Pool using an interface called as Poolable. So this is the Poolable interface. So we are saying that any class which implements this interface(i.e, Poolable) can be pooled in our Object Pool. So what is the contract of this Poolable interface? Well, there is only one method that we're going to add called as reset() method. And the contract here is that we want all the implementing classes to reset their state when this method is called. And this is to ensure that there is no leftover state values that will affect the client. So if we pool the object and if it has some leftover state values, for example, if we have a location that has some value, and if an object gets it and tries to draw it, it will draw at the old location value. So reset is a way that our Object Pool can ensure that the state leftover state is reset. But now it's time to have our Bitmap class implement this Poolable interface. So instead of having our Bitmap implemented, we are going to make our Image interface extend from the Poolable interface so that all images will become Poolable. In the Bitmap class, now we have to provide an implementation for this reset() method. Let's see what we need to reset in our Bitmap. So we have two state values here. We have a 'location' and a 'name'. We know that the name represents a file name and that won't be changed by the client. We only have getter/setter methods for 'location' state that allow the client to read the 'location' state value and change the 'location' state value, and that are related to 'location'. So in our reset implementation, I'm going to simply say 'location=null'. So that takes care of resetting our state. And then I'm simply going to print out a message saying that the 'Bitmap is reset' so that we get some indication that our reset operation was called. Now let's jump into the ObjectPool class. Okay, So this is where we are going to implement our Object Pool. So I have created an empty class here and this class is going to pool anything which extends from Poolable. So we are going to write it(i.e., ObjectPool class) in a generic fashion so that it is more reusable. So first order of business while creating an Object Pool is to provide a way to cache objects in memory. To simplify things, what we are going to do is we are going to use a collection that is provided by Java and the interface is BlockingQueue. So BlockingQueue is an interface that is provided in java.util.concurrent package of Java. We are going to use the generic variable(i.e., T) while declaring the reference variable of BlockingQueue. So this reference variable(i.e., availablePool) is going to represent our in-memory cache. So BlockingQueue allows some level of thread safety, so we can get away with using only the BlockingQueue. Now let's write the constructor for our ObjectPool. In the constructor we are going to initialize our in-memory cache and we are going to use the LinkedBlockingQueue implementation of the BlockingQueue interface. Okay, so this implementation will allow us some level of thread safety when we are adding and removing objects from this queue.

Now we need to provide methods that allows someone to get objects from the Object Pool. So we're going to declare a method called as get(), and this method is simply going to try to take the object from the available pool and return it. Now, this method(i.e., take() method from LinkedBlockingQueue) throws an InterruptedException because it(i.e., thread) is going to block/wait if the queue is empty. So if the object pool is empty and if we're trying to take an object out of it, it's going to block/wait until some other thread put an object in the pool. And if it(i.e., thread) is blocked and somebody interrupts that blocking, it is going to throw an InterruptedException, so in the catch block, we're simply going to print a message on the error stream using System.err.println() and in that case, we're going to say return null. Now, you have to take a decision here. We are simply waiting indefinitely if our object pool is empty. But in real life example, you have to decide what happens if the in-memory cache of the objects is empty. You may want to create new objects if the pool is empty and then add those objects in the pool and then return OR if the object represents an external resource which is limited, for example, like a database connection which is limited by, let's say, licensing requirements or sockets which is limited by operating system, in those cases, you may want to wait if you have reached the limit that is allowed by the external resource. However, for now, we are simply going to wait until some object becomes available to us. So this is the get() method and using this method, somebody can ask for the object from this pool. Now, the next operation that we need to do, inside ObjectPool class, is the release() operation. So when somebody asks for an object from the object pool, we will give it. They will use it for some time and they must return that object to our object pool so that we can add it back in our in-memory cache. And to do that, we have the release() method. So in this method, the most important job is to call the reset() method so that the object state is now clean and it is reusable so that we don't have to do anything in the get() method. Once our reset operation is completed, then we're going to put the object that is given to us into the pool. So this put() method of LinkedBlockingQueue class again throws an InterruptedException. So, we will surround that with try-catch. Now, we have our Object Pool ready. We have a get() operation. We have a release() operation and we have in-memory cache.

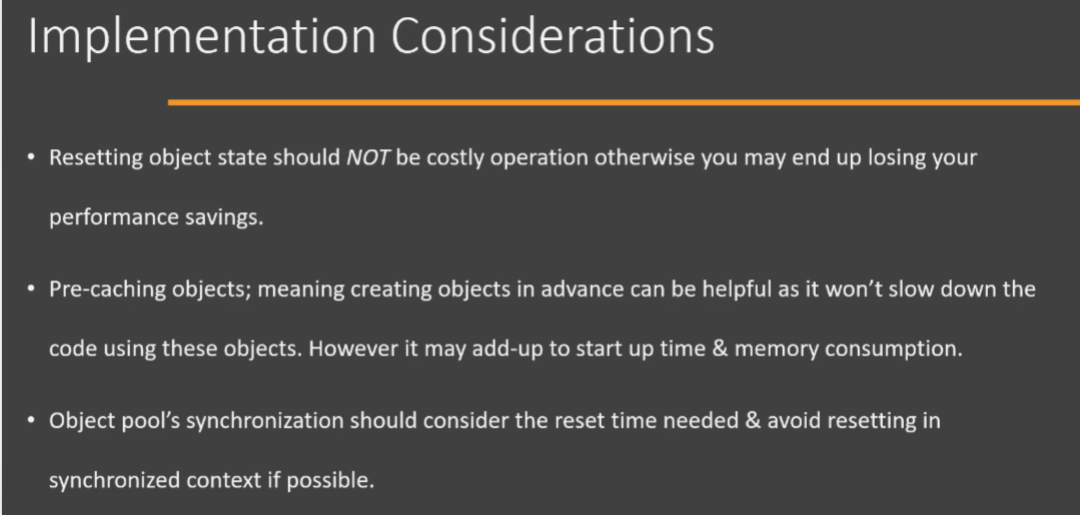
Now, how objects are going to be created for the first time in this pool? So in this example, we are going to create some objects when our object pool is created. So for that, I declared two parameters in our ObjectPool constructor. The first parameter is the Supplier, which is a functional interface, and it has a simple method called as get(). So we can use the 'creator' parameter, which is an implementation of Supplier interface, and get object of Bitmap. And the 'count' parameter represents how many objects we are going to create when we are initializing our ObjectPool. So once that is done, let's loop over. So we're going to write a simple loop where i=0; i<count; i++. In each iteration, we are going to create an object of Bitmap and add it to the pool. Notice, we have used offer() method of BlockingQueue because it doesn't throw any exception. So we're going to pre-create some objects in this implementation. Now our Object Pool has been implemented fully.

Now let's jump over to the Client class and let's try to use this Object Pool. So first of all, I'm going to create an Object Pool and this Object Pool is going to be for Bitmap. So this will be your static final variable that will represent our Object Pool and then we have to provide two arguments to our ObjectPool's constructor. First, we need to supply the Supplier interface implementation and then we have to provide the number of objects that we want to pre-create. Since, Supplier is a functional interface, we can pass the Lambda expression as the implementation. Now let's use our Object Pool to get some Bitmap objects out of it. After getting the Bitmap object, let's change its location. So I'm going to say setLocation() on the Bitmap object and let's use a Point2D class which is from the Java library(i.e., javafx.geometry package). Let's get another object(i.e., b2 reference variable) from our Object Pool and change its location as well using the setLocation() method. And then we can call the draw() method/operation on these Bitmap objects which we got from the Object Pool. So this draw operation is not going to actually draw any bitmap. It's simply going to print some messages on the console. And finally, we need to release these objects(i.e., b1 and b2) to the Object Pool. So we're going to say bitmapPool.release() and we're going to pass the Bitmap object that we want to release. So this is how you are going to use your Object Pool. So once the Object Pool is created and there are some objects in it, we can simply call this get() method to get hold of the pooled object. We can use that object as a regular object. We can call any methods that we want on that object, like we called setLocation() method in our example. And once we are done, our client's job is to release those objects back into the pool so that they can be reused.

So this is how you can implement the Object Pool and use it. So you create some in-memory cache using any data structure, you can ask for method or a way to create initial objects in the pool. And then you apply these methods[i.e., get() and release()] that allow anyone to get the objects from pool and release them back into the pool. So this is a simple implementation of Object Pool design pattern.

1. Object Pool – Implementation & Design Considerations:

Now let's consider some of the implementation and design considerations while using Object Pool. First point is that the reset operation that we call on the pooled object when we are adding it back into the pool, it should not be a costly operation, otherwise we may end up losing all of our performance savings. Now we can pre-cache objects in our object pool. That means as soon as our object pool is created, we are going to create some fixed number of objects and add them into the pool so that when someone asks for that object for the first time, these objects can be easily given to the calling code. Now this improves the performance because now the call to the object pool is not going to take a long time. However, you should keep in mind that doing so, that is pre-creating the objects can add to your startup time and it can also increase your application's memory consumption because you are always going to have these many objects in memory and they will take up some memory or external resources. So you should keep a consideration of this point as well. Next, the synchronization that we implement in the Object Pool, it should take into consideration where we are actually doing the reset operation on the object. Now if you have an object, that is pooled, which needs a lot of time for completing its reset operation, then you should try to move that reset operation out of your synchronized context because now you are blocking your Object Pool or maybe some internal data structure, just to reset a returned object. So if your reset operation is a costly operation, you should do it outside of synchronized context.



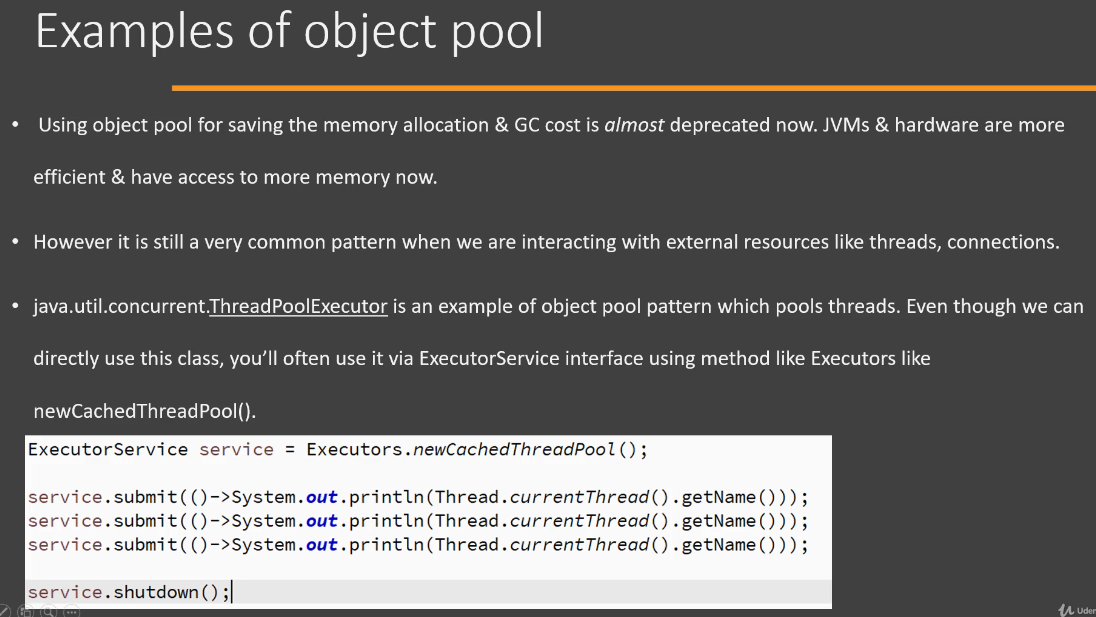
Now one thing that you can do with Object Pool is that we had this get() method that the client calls in order to get a pooled object from our Object Pool. Now, this method can be parameterized. That means we can ask the client to pass some argument and that will be our criteria. And based on that criteria, we can return different types of objects. So this way you can use a single object pool to cache objects of multiple classes, and the get method will provide the criteria which determines which object should be returned from the pool. Now, pooling objects is only beneficial if it involves costly initialization because of some external resource like a connection(i.e., database connection, etc), a socket or a thread, or any other external resource. Do not pool objects just to save on some memory. Unless, of course, you are running into out of memory errors and you have a object initialization that takes a lot of memory for creating an object. If you are in that situation, which is very rare, then it's fine. Otherwise, the only situation where an Object Pool can really improve the performance is if the objects, that are pooled, are tied to some external resource, like database connection which is limited, because then you have some guidelines on how large your object pool can grow or how many objects you need to cache. And since these resources like connection, thread and socket are external to our JVM, initialization of any of these is going to be a costly operation because that will involve a native call to the different operating system and it may depend on the external factors which are not in our control. So in those situations it makes perfect sense to use an object pool so that you acquire that resource and keep it ready in your pool to be used. But don't pool any object or any class's Object just to save onto some memory. Next point is that you should not pool any long-lived objects. And by long lived, I mean objects which are used by client or which are in use by the client for longer durations. Because what is going to happen is that once the client gets hold of these objects, they are not going to return to your pool for a long time. So that particular object is a bad candidate for pooling because your pool is going to keep on creating new objects when somebody asks for it or it's going to block that thread when somebody asks for such an object. So try to not pool such objects which are in use for longer duration by the client.

A black and white text on a black background

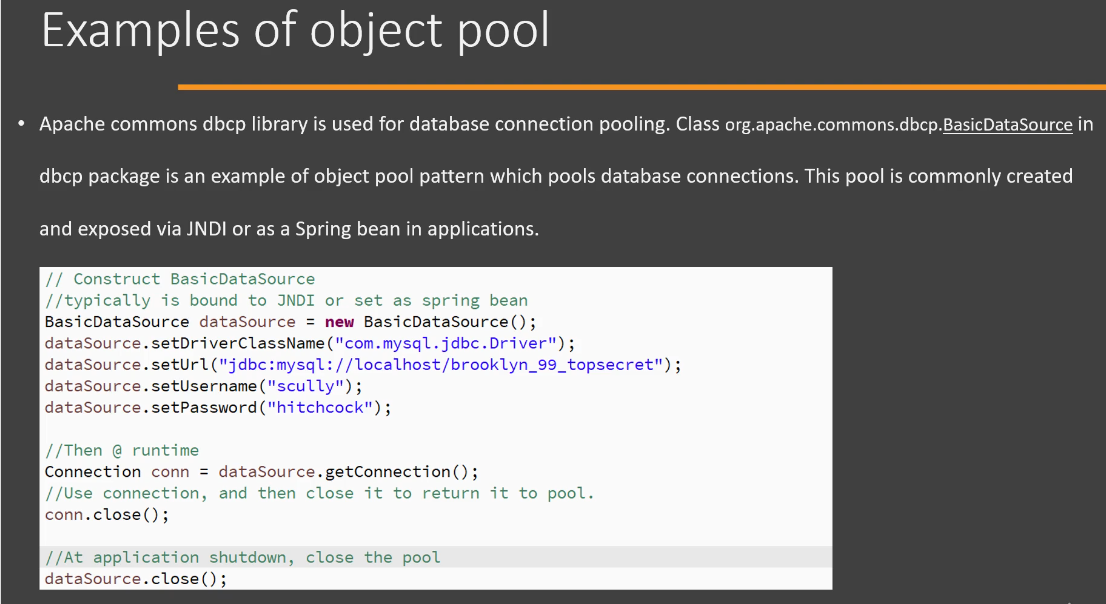
Description automatically generated

1. Object Pool – Real world Examples:

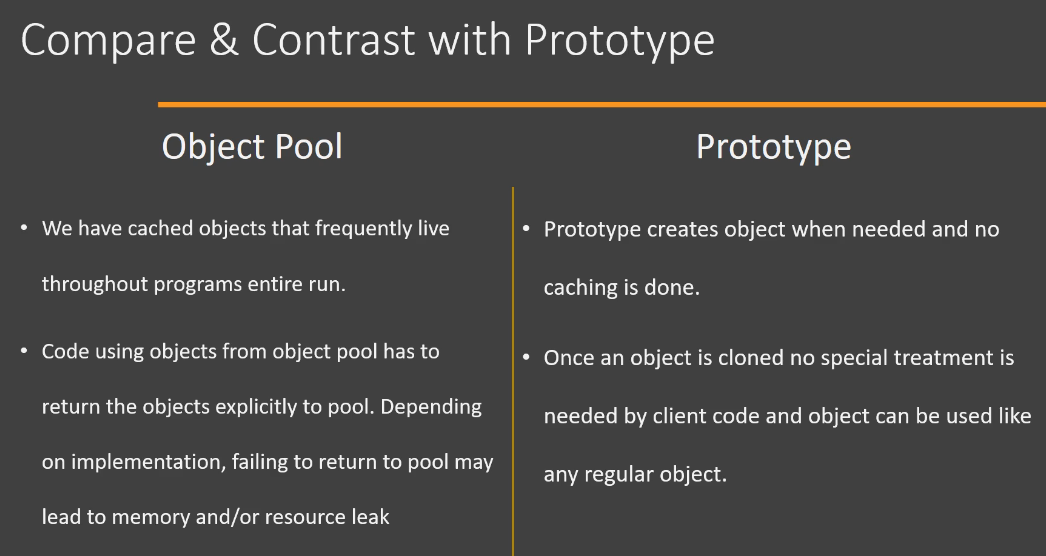
Now let's look at some examples of Object Pool. So before we begin, I would like to highlight again that we do not use Object Pool for saving some memory or GC cost. That type of usage of Object Pool is now almost deprecated because we now have more performant hardware and we have access to more memory. However, Object Pool is still a pretty common design pattern whenever we are interacting with external resources like threads and connections as we have already discussed. So let's look at an example of Object Pool. So one example that you will find in the Java class library itself is the class ThreadPoolExecutor. Now this class is part of the java.util.concurrent package in the Java class library, and it is an example of Object Pool where we pool threads. So you can call that ThreadPoolExecutor as a thread pool. So even though we can directly use this class, we will often use this class(i.e., ThreadPoolExecutor) indirectly whenever you are creating ExecutorService using the Executors class. So Executors has some static methods[for example, newCachedThreadPool()] that allow us to create different types of ExecutorService instances. So this ExecutorService implementations behind the scenes use the ThreadPoolExecutor. So below is a code snippet where we are using the Executors class and calling the newCachedThreadPool() method. So this implementation of ExecutorService uses our ThreadPoolExecutor class behind the scenes, and whenever we submit a Runnable to this ExecutorService, it will create a new thread if it is not present in the pool and execute our Runnable on that thread. And once that is done it will cache or pool that thread in the thread pool. So this is an example of Object Pool that you can find in the Java class library.



Now another example of Object Pool, which is very common, is actually the database connection pooling. So we have a Apache Commons DBCP library which is provided by Apache Commons project and we use this library for database connection pooling and the class that is of importance to us, whenever we are discussing Object Pool, is the BasicDataSource class, so the actual/full class name is 'org.apache.commons.dbcp.BasicDataSource'. So this class acts as an Object Pool and it pools the database connections that we are making. And you will find that this DBCP connection pool is often created and configured using JNDI or it is configured as a Spring bean. So here/below is a simple example where we are actually creating the DataSource/BasicDataSource object in the code itself instead of creating it via configuration. But it will highlight some important points. First of all, we configure our BasicDataSource with the properties that are needed to connect to a database server, and once that configuration is completed, we simply ask for a Connection on that dataSource object. Once we get a Connection back, we can perform any operation using that Connection. For example, we can execute some SQL queries or hand over that Connection to something like JPA. Once our work is done, we close the Connection and this close call[i.e., conn.close()] will actually return the Connection back to the pool where it will decide whether it wants to actually close the connection or simply flush out the transaction and keep that connection in the pool itself. So this is another example where Object Pool design pattern is used for pooling database connections.

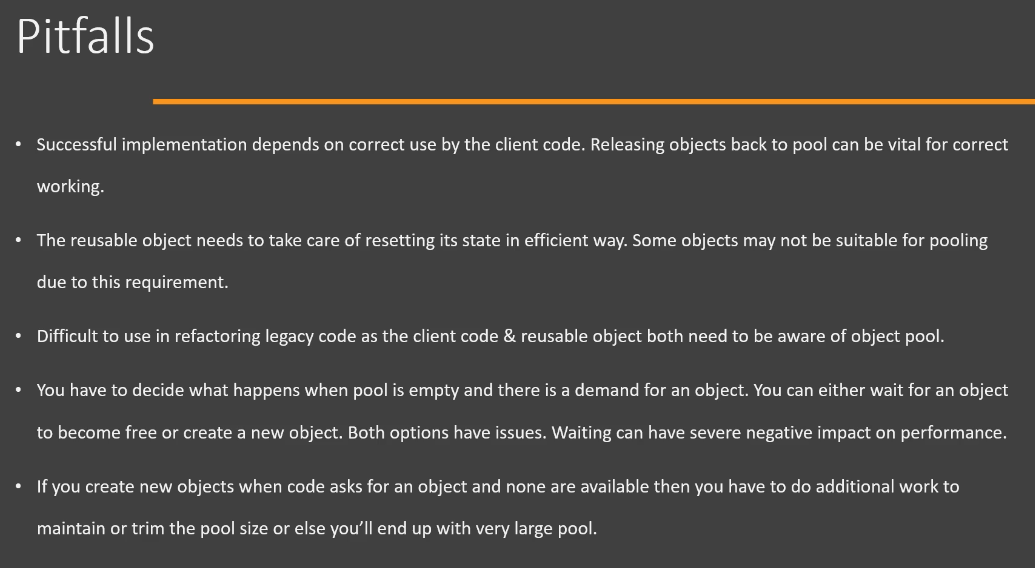


Now let's look at the differences of our Object Pool design pattern with Prototype design pattern. The first difference is that in Object Pool, we cache the objects and these objects can live throughout the program's entire run since they are cached in our Object Pool. The prototype, however, creates objects when they are actually needed and no caching is done by the Prototype design pattern. We simply call the clone() method on our prototype object and a clone or copy of that object is returned to us. That object is never cached by the prototype design pattern itself. Next, the code which is using our Object Pool has an obligation to return the object to the pool and depending upon the implementation of our Object Pool and the pooled object itself, failing to return that object to pool can lead to memory or resource leak. In case of Prototype, however, once an object is cloned, no special treatment is needed by the client code. So client code can keep using that prototype copy as if it is a regular object. So these are a couple of differences between Object Pool and Prototype design pattern.



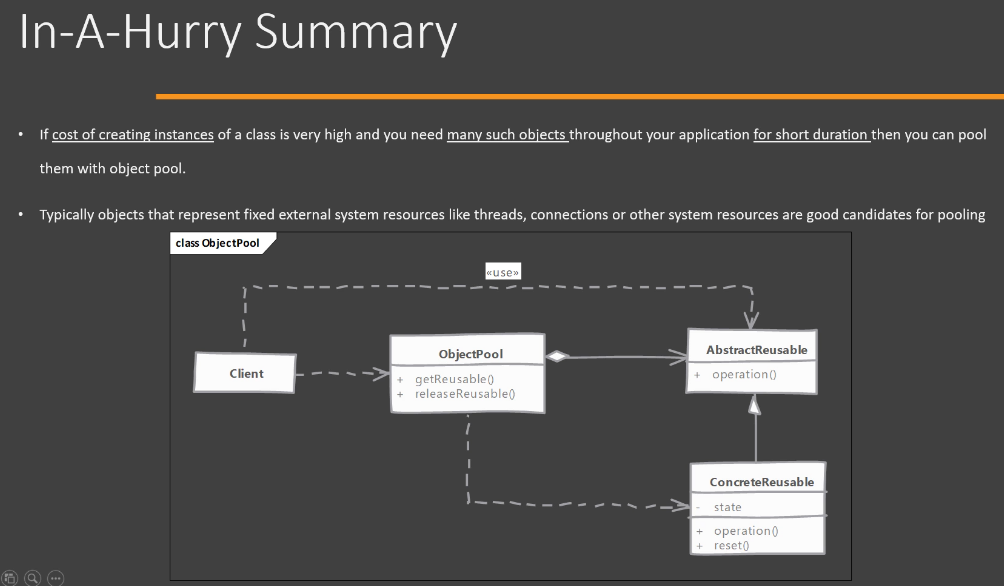
1. Object Pool – Pitfalls:

It's time to learn about pitfalls when using Object Pool design pattern. A successful implementation of this design pattern depends upon the correct use of our pool by the client code. And this code(i.e., client code) may not be in your control because releasing the objects back into the pool is vital for your Object Pool to work correctly. Next, the reusable object itself needs to reset its state, and it must do that in an efficient manner. That means it should not take a long time in order to reset its state. And this is the reason that some objects in your application may not be a good candidate for pooling because they cannot reset themselves in efficient manner. Next, it is difficult to use an Object Pool when working with legacy code because now your client code as well as the object that you are pooling, they both need to be aware of existence of Object Pool because the client needs to interact with your Object Pool and the pooled object needs to provide this reset operation so that its state can be reverted to its default values. Next, you have to decide what happens when your Object Pool runs out of objects and there is a demand for an object. You can either wait for one of the objects to become available or you can create new object. But both of these options can be problematic for you. Waiting for an object can have a severe negative impact on performance, and you may end up with situations where there is a deadlock in your application. And if you keep on creating new objects whenever somebody asks for you and you don't have any object in your pool, then you have to again implement a separate logic that trims your pool from time to time. Otherwise you will end up with large number of objects in your pool. So these are some of the pitfalls when working with Object Pool design pattern.



1. Object Pool – Summary:

Now let's look at the summary of Object Pool design pattern. Now whenever you have a class and creating objects of that class is a costly operation and you need many such objects in your application. But each object is used for only a short duration. Then you can use an Object Pool. Typically objects in your application which represent an external resource like a thread, a class that represents a connection to an external resource like database or JMS or a class that represents something like socket, these are good candidates to be used or to be pooled in an Object Pool. Now let's look at the UML diagram of the Object Pool design pattern. It's quite simple. We have an AbstractReusable, it could be an abstract class or interface, so this is the interface that is used by the Client and this interface(i.e., AbstractReusable) defines the operations that can be performed on that particular object. Then we have ConcreteReusable class and this is the implementation of our AbstractReusable interface. This class(i.e., ConcreteReusable) has its own state and it implements the operations that are defined by the AbstractReusable interface. It also provides an operation/method called as reset() and this is the operation that resets the state of this object(i.e., object of ConcreteReusable class) into its default value. And this reset operation is called by the ObjectPool so that whenever it provides this pooled object to the Client, it can be used without any problems. Next, we have our ObjectPool class. Now this class will be caching objects of our ConcreteReusable class. There are only two methods that are really important here. One method is called as getResuable() method, and this method will return a pooled object from our internal cache. Then there is another method[i.e., releaseReusable()] which is called by the Client when it wants to release an unused object. So these are the two methods which are used by our Client to get the pooled objects in and out from this class.



Now, the objects that we are going to pool, they should provide this reset() method that we just discussed. Now, one point to remember is that this reset operation should be efficient. Otherwise, the release() call, which the client is making, is going to take a lot of time and it's going to add to the performance cost. Next, the pool must handle synchronization issues effectively so that your pool can be used by realizing some performance gains. Next, the reset operation that we just discussed, it should be called or it should be invoked before we add that object into the pool. Now the client code has a responsibility that it must release unused objects back into the pool. And a failure to do that from the client code can break your entire system because you may be leaking resources because of that. Some pools like thread pool(i.e., ThreadPoolExecutor) can handle these situations on their own because a thread can determine when it is done with its work, so it can return itself back into the pool. But many other classes don't have this luxury, and client has the responsibility to explicitly release these objects back into the pool. Next, object pools are difficult to optimize because they are very sensitive to system load, which they face at runtime, so you should keep that in mind. Pools are a good choice when the object that is being pooled represents a fixed quantity of externally available resource, like, for example, thread or connection or the sockets that we can open. Lastly, if you create objects when the pool is empty, then we have to also write code that makes sure that we are trimming our pool back to a reasonable size. Otherwise we may end up with a large number of pooled objects in our memory.

A screenshot of a computer

Description automatically generated

**Note:**

So we have this interface called as Poolable, and it defines only one method called as reset(). The reason we have this interface is that we can code our ObjectPool using this interface, so that way, any class can implement this Poolable and our ObjectPool can work with that class. Currently, ObjectPool is implemented as a generic class and we are going to work\_with/pool any class who implements the Poolable interface. We are using a BlockingQueue here(i.e., ObjectPool class) but, of course, you can use any data structure that you want. The important point here is that this particular cache(i.e., private BlockingQueue<T> availablePool;) which you are using, it should be able to correctly handle the synchronization issues. Inside this ObjectPool class, we have written a constructor that creates some objects in the pool beforehand. Then we have a get() method inside ObjectPool and this method will simply return a pooled object from our cache and the release() method, which takes an unused object, resets that object state and puts it back in our object pool. Now, you should handle synchronization issues inside these two methods[i.e., get() and release()] and you should make sure that your reset operation is reasonably fast.

When object creation cost is very high and each object is used for a short duration as well as large number of such objects are needed, Object Pool can improve application performance. Object Pool will call the reset method on the object before returning it to the cache. One of the best use case where Object Pools can help is if an object represents an external resource which is limited, like database connections.