

8. Using many instances of an object while keeping code running smoothly

Imagine that you have a requirement to use many similar objects in your application. Perhaps, you are building a distributed application based on microservices architecture and each of these objects represents a unique connection to one of your service instances. Or maybe you are interacting with multiple database entries and, for each one of them, you need an object that represents a database connection.

In this situation, purely using inbuilt language features will probably be problematic. You will need to instantiate every single one of these objects. Once the object is out of scope, your runtime will need to get rid of it to free up the memory. Once you need a similar object again, you will instantiate it again. And so on.

If you follow this approach, you will probably experience a performance hit. Instantiating a new object each time you need to use it is a relatively expensive process. The runtime will need to allocate memory for it and populate those chunks of memory with new values.

If you are using similar objects in different parts of the application, you will be required to allocate sufficient memory to each instance of the object. And that may become quite a lot of memory if you need to use many of such instances.

Likewise, when you are instantiating a new object, there is always

a cost associated with running the constructor of the object's data type. The more complex the constructor logic is, the bigger performance hit you will get.

Finally, when you are no longer using an instance of the object and it gets out of scope, there will be a performance cost associated with garbage collection. Remember that the memory will not be freed straight away. It will still be occupied until the garbage collector has found your object and identified that it is no longer being referenced anywhere.

The latter, of course, doesn't apply to the languages that don't have an in-built garbage collector. However, in this case, you will have to free the memory yourself if you don't want to introduce a memory leak. So, if you are using one of such languages, like C++ or Rust, you will have an additional issue in your code to worry about.

Suitable design patterns

Object Pool

Object pool is a pattern that allows you to reuse the object instances, so you won't have to keep instantiating new objects every time.

You have a single Pool object that stores multiple instances of instantiated objects of a specific type. If any of these objects aren't being used, they are stored inside the Pool. Once something in the code requests an object, it becomes unavailable to any other parts of the code. Once the caller has finished with the object, it gets returned to the Pool, so it can be reused by other callers.

Initially, you will still need to instantiate objects in the Pool. However, when your Pool grows to a reasonable size, you will be instantiating new objects less and less, as there will be a greater chance of finding objects in it that have already been released by their respective callers.

To ensure that the Pool doesn't grow too big, there will be a property that will limit its size. Likewise, a mechanism inside of it will determine which object instances to get rid of and which ones to keep. For example, you don't need an object pool with 1,000 object instances if your application will only ever use 10.

Why would you want to use Object Pool

- Objects are being reused, so you will not get any performance penalty associated with instantiating new objects.
- Object pool maintains its size as needed, so you will not end up with way more object instances than you would ever expect to use.

Flyweight

This design pattern allows multiple objects to share parts of their state. So, for example, if you have a thousand objects, all of which currently have the same values in some of their attributes, this set of attributes will be moved to a separate object and all thousand instances will be referring to the same instance of it.

This allows you to store way more objects in the memory than you would have been able to otherwise. In the case above, instead of having a thousand instances of a particular object type with all their property values in each, you would have a thousand basic skeleton objects, each of which occupies only a tiny amount of space in memory. The remaining properties of these objects will be stored in memory only once.

One disadvantage of flyweight, however, is that it makes your code complicated. You will need to decide which parts of the state are shared and which aren't. Also, you will need to do some thinking on the best way of changing the state once it becomes irrelevant to any specific instance of an object.

Based on this, it's recommended to only use flyweight if you absolutely must support systems where the performance of your code would very noticeably decrease otherwise.

Why would you want to use Flyweight

- Squeezing way more information into memory than you would have been able to otherwise.

Prototype

Prototype, which we have already covered in [chapter 7](#), can also help to solve this problem. However, unlike Object Pool, it will not give you any performance benefits.

What Prototype will give you in this situation is the ability to create many similar objects without having to go through a complex process of defining their field values each time.

For example, you may have a service that communicates with several instances of the same microservice, and each of these instances is presented by an object in the code. Let's say that all microservices are accessed by the same IP address, but a different port. The rest of the connection parameters are also identical.

In this case, your configuration file may contain all shared connection parameters. And you will only have to go through the configuration once to create the prototype object. After that, to create any new connection objects, all you'll have to do is clone the existing one and change the port number accordingly.

To gain performance benefits, Prototype can be combined with Object Pool. Prototype is especially useful when the objects in the Object Pool are complex.

8. Using many instances of an object while keeping code running smoothly 103

Why would you use Prototype alongside Object Pool

- The objects in the Pool are much easier to instantiate, as they can now be cloned.