**Kotlin multithreading: Comparing .wait(), .sleep(), and .delay()**

Author: Ivan Garza

Date: April 12, 2023

Multithreading programming is one of those taboo subjects that not many developers enjoy talking about. The complexity of the topic has driven away even experienced programmers for as long as it has been helping us solve problems.

This had been particularly true for the Android platform, especially back when we were predominantly using Java. Between the juggling of different threads, its Handlers and Runners, there was little time for sanity. And let’s not mention the memory leaks.

When the Android OS was introduced to the world of coroutines by the new programming language Kotlin, everything changed. The practice of multithreading programming stopped feeling as intimidating or foreign as it did before. Kotlin and its coroutines allowed for developers to write multithreading code in a cleaner and more efficient way. Check out this article to learn more information on what Kotlin coroutines are and how to use them.

Kotlin coroutines introduced a new set of keywords and APIs that made multithreading patterns easier to read and ultimately understand. Keywords such as suspend and runBlocking allow for a more human-readable way of understanding what these multithreading APIs do, and allow devs to better establish their expectations. For a deeper explanation on the two keywords mentioned above and their usage, check out this article.

In this article, I will explore three individual functions that are commonly used in multithreading programming while working in Kotlin: wait(), sleep(), and delay(). As you may have noticed, not all of these three functions actually belong to the Kotlin programming language, but instead we’ll approach this inquiry into the particularities of the multithreading paradigm from a broader perspective.

**Simplify your codebase with Swift’s decorator design pattern**

Author: Yusuf Ahmed

Date: March 29, 2023

When developing software, keeping a clean and manageable codebase is essential for efficient and sustainable growth. However, as a project’s complexity increases, it can become challenging to maintain a readable codebase. Here, the decorator design pattern comes into play.

In this article, we’ll delve into the power of decorator design patterns in Swift, learning how they can simplify our codebase and improve its structure. We’ll cover the basics of the pattern, including its definition, purpose, pros and cons, and how to effectively implement it in a project. Let’s get started!

**What is Swift’s decorator pattern?**

The decorator pattern, also called the wrapper pattern, is a structural design pattern in object-oriented programming that utilizes composition, providing a flexible way to dynamically extend an object’s behavior without affecting the behavior of other objects of the same class.

The decorator pattern can also act as an alternative to the traditional method of adding behaviors through inheritance, where a new subclass is created by inheriting from an existing class.

**When to use the decorator pattern**

You should use the decorator pattern when your project’s requirements call for dynamic modification of an object’s behavior, thereby making subclassing an inflexible solution.

For example, you should use the decorator pattern when you want to:

* Add new functionality to an object at runtime without affecting the behavior of other objects of the same class
* Create variations of an object with different combinations of behaviors and avoid creating many complex subclasses
* Customize components in an application without affecting the behavior of other parts of the system
* Avoid the limitations of inheritance, like tight coupling and inflexibility, and instead utilize composition for more modular and maintainable code

**How to implement the decorator pattern in Swift**

Now that we’ve covered the basics of the decorator pattern and when to use it, let’s take a closer look at how to implement it in Swift.

We’ll use an example of a pizza ordering system where customers can customize their pizza by choosing different toppings and crust types. In this scenario, we can use the decorator pattern to allow for dynamic modification of the Pizza object’s behavior based on the customer’s choices.

**Using Cow in Rust for efficient memory utilization**

Author: Yashodhan Joshi

Date: March 22, 2023

Clone-on-write — or Cow for short — is a convenient wrapper that allows us to express the idea of optional ownership in Rust. In this article, we will learn what it exactly is, what problem it aims to solve, and how we can use it in our Rust code.

**The problem with occasional mutation**

Consider a case where you have a Vec of some elements. For the purposes of this example, consider that an Element has an ID that can uniquely identify it, and is large enough in size that duplicating it would require considerable memory.

In this case, we need to make sure the vector only has unique elements and does not contain any duplicates. To ensure this, we have a function that can filter the Vec.

As you can see, this simply keeps the IDs in a HashSet. When a new ID is encountered, it pushes the element into the Vec we return.

Let’s say we are expecting the input in our use case to frequently contain duplicates, or we know for sure that the input will always contain duplicates. In that case, we must create a temporary Vec, input all the elements whose IDs we have not seen before, and finally return this as the Vec of unique elements.

In situations where duplicates are common, this implementation is pretty efficient. But what if they are not so common?

When the input only infrequently contains duplicates, we would be creating the needless temporary Vec far more often than actually necessary. As a result, we would also be unnecessarily allocating that much extra memory and spending CPU time copying the elements over from the original to the new.

As we know, an Element is big in size, so copying it takes considerable memory. We also need to allocate and grow that much memory for the returned Vec.

Even if we took the input by value, not as a reference — in which case we would not have an overhead of calling clone on the elements — we would still need to allocate extra memory for the returned Vec and then free up the original after we are done copying.

This seems wasteful, considering that when there are no duplicates, we are essentially simply returning the input array.

Cases like this involve performing some costly operation on the input, but only when a certain condition is true. When that condition is so rare that the output is simply the input most of the time, Cow can help us to avoid the cost of the operation.

**Writing interactive component stories with Histoire in Vue 3 and TypeScript**

Author: Sebastian Weber

Date: March 15, 2023

With the release of Vue 3, a Storybook alternative named Histoire has gained some traction. In this blog post, we’ll explore how to create interactive documentation of UI components for Vue 3 projects. As with Storybook, Histoire is designed for making your life much easier to create living style-guides and component showcases.

In contrast to the previous LogRocket article that gave an overview of Histoire’s concepts and features (e.g., Histoire’s stories or variants), the focus here is on the development of Histoire’s UI controls (e.g., HstText as a text input field or HstCheckbox as a checkbox field) as you can see on the right side of the below screenshot. You can consider this article as a sequel to my earlier LogRocket article on Storybook controls for React projects.

**Our companion project**

You can follow along with a companion project that features an implementation of todomvc with Vue 3 and TypeScript. We use Histoire to illustrate different development patterns, such as how to initialize the Pinia store for a Histoire story that relies on it.

**Histoire’s UI controls**

Histoire is a tool that allows you to document and test UI components by providing a way to interact with them. This is where controls come into play. The controls shown in the controls panel are form fields that enable users to interact with UI components rendered in the preview canvas.

As an example, in our companion project, we use a HstSlider component that represents a slider control, allowing the user to adjust a value by sliding a handle along a track. It can be used to adjust numerical values in a range. Thereby, the values of the previewed UI components get updated.