Deep dive into Asynchronous Programming or what Coroutines, Promises and Reactive Observables are and how they are implemented

Arseny Savchenko

Innopolis University

13 May, 2023

Contents

Chapter 1

Introduction

Asynchronous programming, being a powerful tool for parallel execution of several tasks, has become one of the main features of modern programming languages such as Kotlin, Rust, Go, Dart and others. Concepts of "async/await" and reactive programming help to manage enormous number of tasks that require waiting until completion (eg. database access, network request, file reading, image processing, etc.).

Coroutines, promises and reactive observables are widely used in many spheres: from client UI development (such as Frontend development or development of mobile client applications) and game development to highly loaded servers (eg. "RxJava"l "WebFlux" of Spring Framework, "Flow" of Kotlin and "Ktor" framework, "Tokio" Framework of Rust, etc.).

The most powerful and beloved idea of asynchronous programming is that it is very cheap. For instance, application will hardly be able to manage hundreds of simultaneously-running threads. At the same time, coroutines can be launched by the millions.

What lies behind such a strong concept and how to organize asynchronous work, even if the programming language does not support the concept of async/await or reactive programming? This research will reveal the veil of secrecy of coroutine management and the implementation of reactive programming.

I Methods

Understanding of the concepts of asynchronous approach requires knowledge of implementation of various mainstream libraries and frameworks for asynchronous programming with concepts of async/await and reactive programming. During the observation of the source code and practical usage of different libraries, all of them were split into groups according to 1) language 2) model of asynchronicity 3) similarities of implementation and usage 4) cases and spheres where those libraries are used. With this approach libraries and frameworks can be trivially compared and contrasted by criteria above.

II Literature Review

Research is mostly focused on the details of how coroutines and observers are managed and implemented, that is why most part of sources are based on official documentation or code samples from various frameworks and libraries used in different programming languages. Tutorials and scientific articles from professional developers also have to be considered in order to understand practical usage of asynchronous approach as well as best practices.

In order to deeply understand concepts of asynchronous programming, sources were divided into two groups by topics: 1) Reactive programming 2) "Async/await" concept, suspending control flow and "Promises" concept. Also sources were separated by programming languages which implements such concepts: 1) Kotlin 2) Rust 3) Go 4) JavaScript/TypeScript.

A. Reactive programming

Kotlin language has various implementations of Reactivity concept. Sources about different Flows from Coroutines were considered as modern approaches to solve problems efficiently. Classic JVM solutions such as RxJava were also considered in order to understand implementation for JVM-based code. Moreover, specific libraries such as LiveData and AsyncTask for Android and others were also studied to demonstrate variety of implementations and purposes.

As for Rust language, study is focused on RxRust library, which presents classic solution for modern language with functional paradigm. Moreover, mixes with Tokio and Futures crates were also reviewed to show connections between suspending execution and Observer pattern.

Finally, modern UI libraries with reactive UI state management and concept of "recomposition", such as ReactJS, Compose and Flutter were considered in order to introduce architectural concepts of reactive UI and recomposition principle.

B. Suspending functions and Promises

Study is focusing on concepts of Kotlin's Coroutines. Sources about asynchronous libraries, such as Ktor and SQLDelight, were also reviewed in order to illustrate practical usage of Coroutines.

As for Rust language, asynchronous crates, such as Tokio and Futures, were reviewed in order to understand semantics of implementation and compare it with Kotlin's implementation.

As for Go language, sources about Goroutines were reviewed. Official documentation is examined in order to deeply understand effectiveness of high-performing go-routines and their effect on backend development.

JavaScript/TypeScript as main languages for frontend development, as well as Dart with Flutter cross-platform UI framework, were considered because of their "Promises" concept.

Chapter 2

Idea of Asynchronous Execution

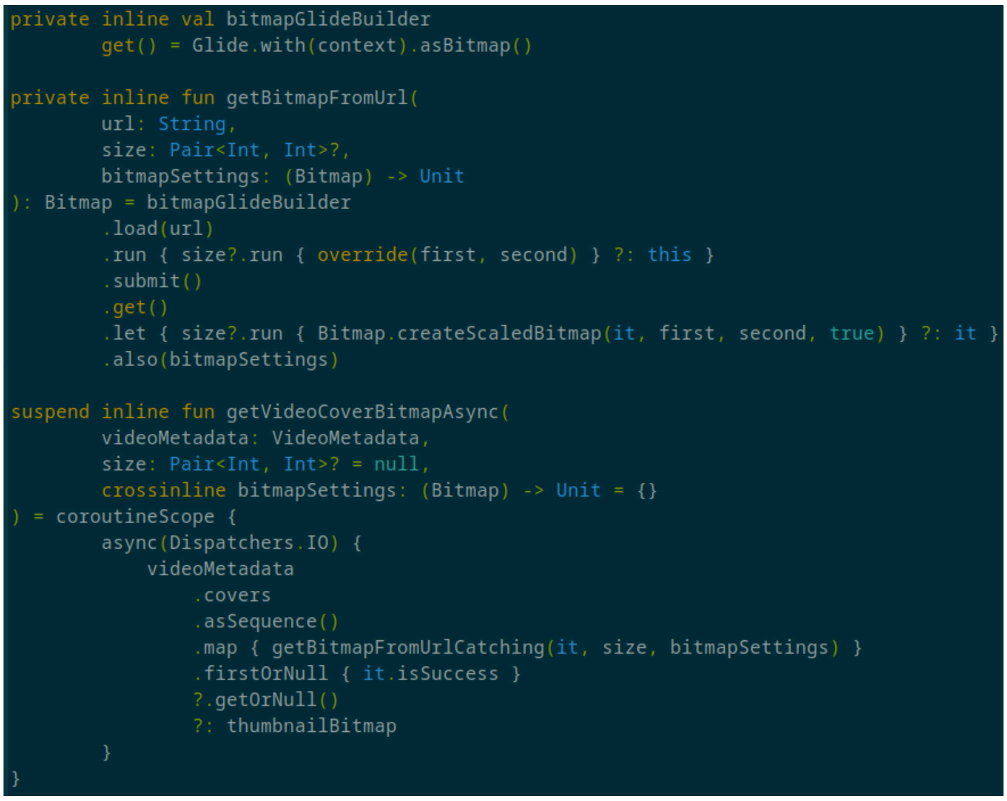
A huge advantage of asynchronous program execution in comparison with multi-threaded is a significant cheapness in terms of device resources. Unlike the whole thread allocated by the Operating System, asynchronous execution modifies the current control flow depending on the workload of the task and implementation method.

I What is Asynchronous Programming?

All appropriate cases to use of asynchronous approach can be described as **smart managing of time with managing control flow activity**. In other words, when thread has multiple tasks to complete, but it is stuck with simple waiting, which is not smart distribution of time and resources. In other words, **thread is just wasting execution time by not doing useful job**.

Asynchronous approach is intended to illuminate such a wasteful execution. Usually, this method is beneficial in next scenarios:

1. **Handling User Interface (UI) updates**. Since UI has to be modified in main thread, it is not easy to access, transform and show some complicated parts of data, such as images and videos. Besides, competent distribution of resources with combination of threads may solve performance issues for application with deeply-detailed UI. For example, next code sample illustrates image processing with Coroutines and non-coroutines based Glide library for Android. Fig. 2.1 demonstrates this example.



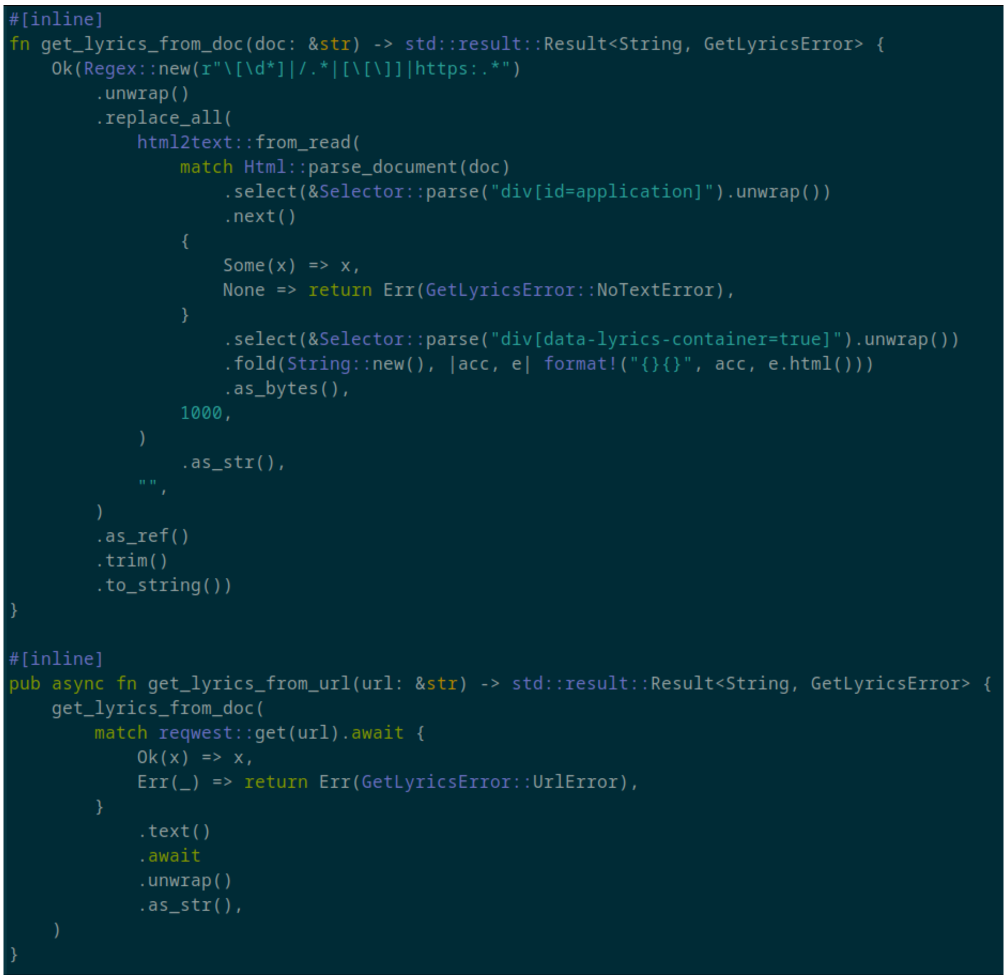
```
private inline val bitmapGlideBuilder
        get() = Glide.with(context).asBitmap()

private inline fun getBitmapFromUrl(
        url: String,
        size: Pair<Int, Int>?,
        bitmapSettings: (Bitmap) -> Unit
): Bitmap = bitmapGlideBuilder
        .load(url)
        .run { size?.run { override(first, second) } ?: this }
        .submit()
        .get()
        .let { size?.run { Bitmap.createScaledBitmap(it, first, second, true) } ?: it }
        .also(bitmapSettings)

suspend inline fun getVideoCoverBitmapAsync(
        videoMetadata: VideoMetadata,
        size: Pair<Int, Int>? = null,
        crossinline bitmapSettings: (Bitmap) -> Unit = {}
) = coroutineScope {
        async(Dispatchers.IO) {
            videoMetadata
                .covers
                .asSequence()
                .map { getBitmapFromUrlCatching(it, size, bitmapSettings) }
                .firstOrNull { it.isSuccess }
                ?.getOrNull()
                ?: thumbnailBitmap
        }
}
```

Fig. 2.1. Images processing with Coroutines and Glide library

2. **Accessing databases and files**. Files operations are usually managed by Operating System, meaning that it is possible to implement such operations with non-blocking API (e.g. Jetpack Room library for Android SQLight with Coroutines extensions or Tokio's extensions for asynchronous file management).

3. **Managing network requests**. Same as databases/files - libraries usually able to implement non-blocking interface (e.g. Ktor library for Kotlin or Tokio's network API and Hyper crate for Rust). Moreover, reactive programming is highly used in operations with "Reactive Web Services" (e.g. HTTP API with Netty engine (JVM) allows to listen multiple socket connections within single process). For example, next Rust crate helps to access and parse songs' lyrics from Genius API with an asynchronous call. Fig. 2.2 demonstrates network request to Genius API to parse lyrics from HTML page.

In other words, asynchronous approach can be described as **smart time management by manipulating control flow**. However, the main question still holds. How exactly does such a smart distribution of device resources is implemented and managed? How are devices with the Vaughn-Neumann architecture able to execute instructions that does not



```rust
#[inline]
fn get_lyrics_from_doc(doc: &str) -> std::result::Result<String, GetLyricsError> {
    Ok(Regex::new(r"\[\d*]|/.*|[\[\]]|https:.*")
        .unwrap()
        .replace_all(
            html2text::from_read(
                match Html::parse_document(doc)
                    .select(&Selector::parse("div[id=application]").unwrap())
                    .next()
                {
                    Some(x) => x,
                    None => return Err(GetLyricsError::NoTextError),
                }
                    .select(&Selector::parse("div[data-lyrics-container=true]").unwrap())
                    .fold(String::new(), |acc, e| format!("{}{}", acc, e.html()))
                    .as_bytes(),
                1000,
            )
                .as_str(),
            "",
        )
        .as_ref()
        .trim()
        .to_string())
}

#[inline]
pub async fn get_lyrics_from_url(url: &str) -> std::result::Result<String, GetLyricsError> {
    get_lyrics_from_doc(
        match reqwest::get(url).await {
            Ok(x) => x,
            Err(_) => return Err(GetLyricsError::UrlError),
        }
            .text()
            .await
            .unwrap()
            .as_str(),
    )
}
```

Fig. 2.2. Acquiring lyrics from Genius API with asynchronous call of reqwest crate

consistently follow after each other?

II When not to use asynchronous execution?

Before highlighting the main features of any type of asynchronous execution, let us focus on the problems that may arise during asynchronous code execution, that are not intended to be solved with either coroutines, promises or reactive observers:

1. ***High Performance Computing***. Since asynchronous execution is more about managing time and control flow, high loaded tasks with intensive execution, for example, the task to find next prime number, described in article [1] by Roman Elizarov (project lead for Kotlin language), will not get any benefit from execution without any multithreaded schedulers. More details are provided in the ”Suspending execution” section.

2. ***Execution of asynchronous function has no benefit in general non-prepared for async task block of code*** or can even cause **thread blocks and dead locks**. As was noted by Roman Elizarov [1], launching non-coroutine task in coroutine environment will not have any effect, however, if we introduce scheduling, asynchronous execution will make some sense. Most async functions in different languages require some special scopes that allow framework to execute coroutines as two different workers with their own control flow. But if developer cannot create this scope (or it will make no sense), asynchronous approach will be excessive.

3. ***Limited scope and availability of resulting data***. Reactive approach (which mostly relies on "Observer" pattern) allows to access result only in limited asynchronous scope, that will be executed after some time. In other words, it is problematic to acquire data after reactive execution without other observers or some hacks (e.g. using Condition Variable primitive synchronization).

4. ***Testing and Debugging***. Unit testing usually requires preparation to isolate the asynchronous block of code under test or cannot be integrated into an isolated test system at all. Moreover, different implementations of asynchronous execution uses their special error handling systems which are not compatible with standard methods of error handling (like exceptions). For instance, Nick Skelton describes in his article [2] that typical unit-testing methods for LiveData and RxJava's Observer are not compatible with Kotlin's Flow because of different scheduling.

5. ***Deadlocks***. Even though coroutines are not as strong as threads in terms of system activity, they still can block themselves (and even their threads) when multithreading is used. Usually, such issues are not resolved by the compiler, so it is up to developer to track down such bugs. For instance, dead locks may arise when managing SQLight transactions with Coroutine-based dispatchers in Kotlin described in article by Daniel Santiago [3].

6. ***Concurrency***. Sometimes, asynchronous code is not intended to execute in multiple threads (although, it can be done, e.g. with Coroutines and Dispatchers in Kotlin or with Thread Pools in Tokio/Futures creates in Rust language), so **synchronization** (or another blocking operation) **will block thread**, which means that **all asynchronous operations in this particular thread will be blocked**. In other words, thread synchronization and thread-blocking operations may not be suitable for async/await or reactive approaches.

If any of those issues are present in current multithreaded code, asynchronous execution may be an ineligible way to fix issues with parallelism.

Chapter 3

Reactive programming

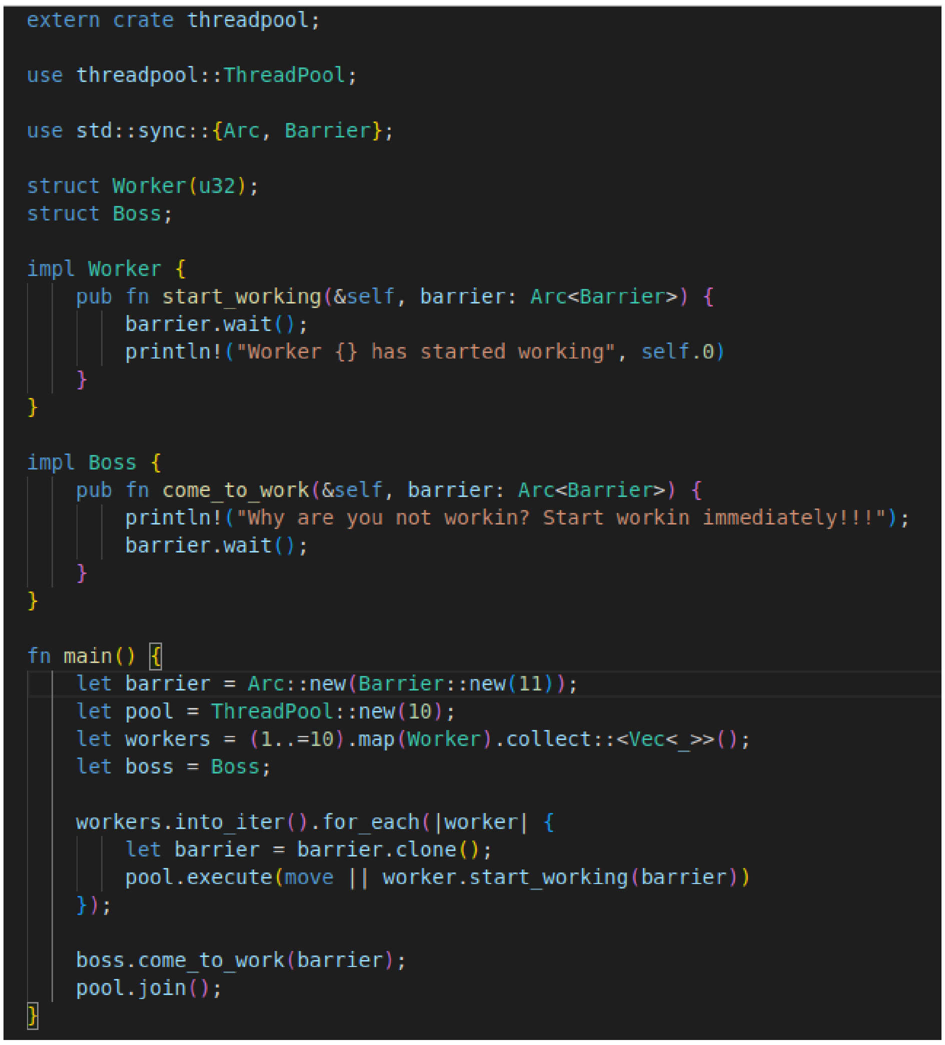
The reactive paradigm (AKA Rx) is a concept that allows you to respond effectively to updates, whether it is changes in data or event stages. It can be described as an "event coding": whenever somethings happens, program reacts on this event (that is why it is called "reactive"). As Keval Patel noticed in his article [4], Rx is a solution to make code reactive to any updates and provide proper reaction to such notifications.

Reactive paradigm improves both user experience and code management. Rx can be chosen for the next reasons:

1. ***Explicit execution*** Result of every call must be easily managed and controlled before it is received. In other words, management of control flow in Rx-based models must be declarative. By calling it declarative, we mean standard functional units, such as filters, combiners (e.g. reduce(), fold(), zip(), etc.) and transformators (e.g. map()).

2. ***Thread scheduling*** Asynchronous calls without scheduling may be useless or can even introduce some errors (such as dead locks). To be able to execute tasks in parallel, scheduler has to be introduced. As a rule, scheduler is based on the concept of "thread pool", described by Kislay Verma in his study [5] in more details. Such dispatching allows to control execution of every operation, dynamically switching threads, if different operations require different threads (e.g. making request from API, then updating UI on main thread).

Thread Pool is a concept, that mostly relates to both ***"fork parallelism"*** and ***"shared mutability"***. It consists of scheduled implicit queue of threads that helps to distribute tasks and sub-tasks on multiple workers. Next code sample demonstrates thread pool parallelism (using threadpool Rust crate) with producer-consumer-like scheduling system. Fig. 3.1 demonstrates idea of Thread Pool with "Barrier" synchronization primitive.



```rust
extern crate threadpool;

use threadpool::ThreadPool;

use std::sync::{Arc, Barrier};

struct Worker(u32);
struct Boss;

impl Worker {
    pub fn start_working(&self, barrier: Arc<Barrier>) {
        barrier.wait();
        println!("Worker {} has started working", self.0)
    }
}

impl Boss {
    pub fn come_to_work(&self, barrier: Arc<Barrier>) {
        println!("Why are you not workin? Start workin immediately!!!");
        barrier.wait();
    }
}

fn main() {
    let barrier = Arc::new(Barrier::new(11));
    let pool = ThreadPool::new(10);
    let workers = (1..=10).map(Worker).collect::<Vec<_>>();
    let boss = Boss;

    workers.into_iter().for_each(|worker| {
        let barrier = barrier.clone();
        pool.execute(move || worker.start_working(barrier))
    });

    boss.come_to_work(barrier);
    pool.join();
}
```

Fig. 3.1. Thread Pool scheduling example

3. ***Maximum independence of execution and minimum side effects*** Ideal case to improve performance and data safety is to completely avoid any data mutability and synchronization, by applying **Fork Parallelism** or **Pipelining**. As was mentioned before, synchronization is usually not the best case for asynchronous execution, so it makes perfect to manage control flow without any data dependency (e.g. using *broadcasting* and *message-passing* techniques).

I Observer Pattern

Before going deep into the details of implementation of Reactive paradigm, reader has to be familiar with the **"Observer"** pattern, described in Gang of Four [6]. According to Gamma et al., the Observer pattern solves the *problem of consistency of actions of objects and their instances by means of timely notification of data updates*

Implementation details may differ within Observers from various libraries, however, all of them share same principles: 1) single class that provides data access and notifies (for example, as a signal with *Message-Passing* interface) if any updates are made to the data; 2) multiple observer that handle data updates. Fig. 3.2 provides UML diagram for the Observer pattern.

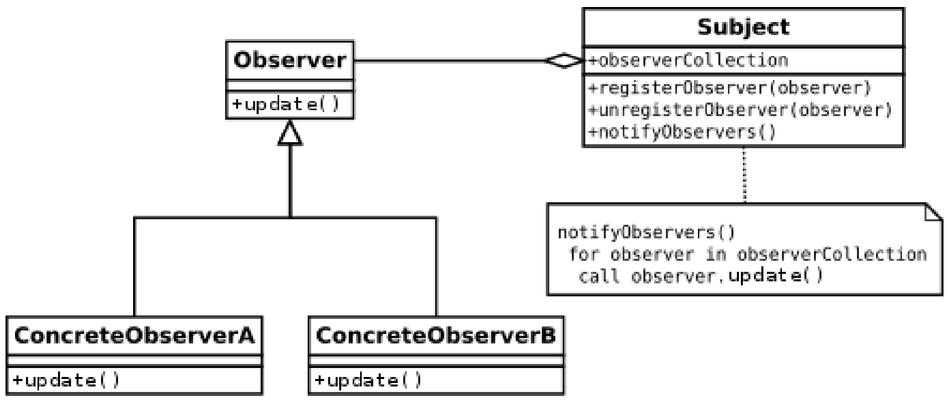


Fig. 3.2. UML diagram of Observer pattern

Nowadays pattern itself is widely used in many spheres: from UI development to High Performance Computing. Its asynchronous nature has evolved to what we now know as "Reactive Programming". Besides classic Rx, it is also commonly used as the way to update whole user interface. For example, such libraries as React, Flutter, Jetpack Compose, Android MVVM + DataBinding, SwiftUI and others use this idea under the hood: they observe all updates to displayable data and launch "recomposition" whenever data was updated.

II Notification mechanisms

Ability to react is not trivial to implement: all in all, it is a core feature of reactive paradigm. Different libraries may use different approaches, but this article will focus on main approaches to implement efficient notification mechanisms.

A. Polling

Polling mechanism is a technique that allows to make a request to OS in order to access data in files or web sockets, while program continues its primary job. Direct requests may be expensive, since all IO operations require some time, which application can spend on other tasks that can be done during the delay (which may take significant amount of time).

Different operating systems support their own ways to establish this mechanism, such as:

- *epoll* for Linux
- *kqueue* for MacOS
- *IOCP* for Windows
- *poll* for Unix-based OS
- */dev/poll* for Solaris-based OS
- *select* for all commonly used OS

Epoll is a system call utility for Linux-based Operating Systems that is able to manage multiple file descriptors at once in non-blocking manner. Kernel scans required file descriptor in order to find updated ones. Then it sends those descriptors back to user. Taking into account implementation of both file system and implementation of network for all Linux distributives, epoll supports both file IO and network IO operations.

The main difference between epoll and other polling mechanisms for Linux-based OS (such as poll and select) is that file descriptors are managed more efficiently by kernel itself. According to the next article about epoll [7], it provides scalability for file descriptors by telling kernel in which file descriptors we are interested.

Kqueue is "epoll" for MacOS. It stores queue of events, that are triggered by file descriptors. Whenever event is happened, user will be notified to properly handle all such cases. As epoll, kqueue supports scalable management of file descriptors. Since MacOS is a Unix-based OS with similar to Linux file system, it also supports both reactive file and network IO management.

IOCP is a similar technique for Windows API. As kqueue and epoll, it uses queue of updated ports that require handling to update. Since file system is different from Unix-based OS, IOCP provides ability to react only for network IO operations.

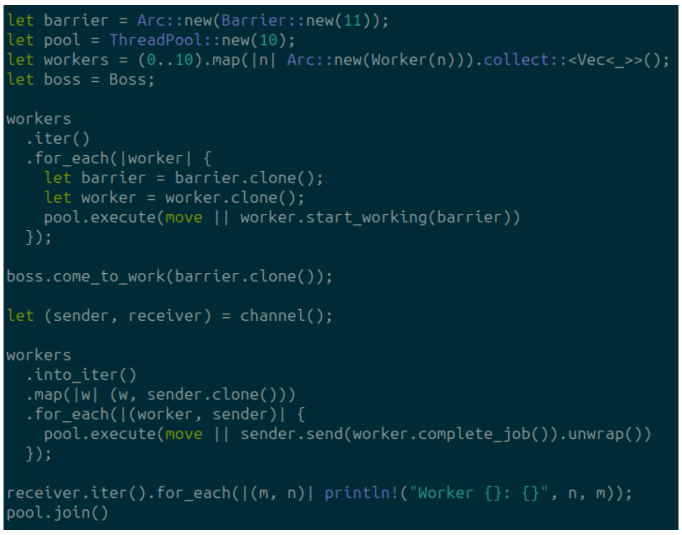
The reason this article is focused on this 3 polling mechanisms and only on those 3 is that they are commonly used in all asynchronous libraries and frameworks with asynchronous and reactive workflow. For example, Rust's Tokio crate for asynchronous IO operations is built upon those polling mechanisms. Besides, it provides extensions for Unix-based and FreeBSD-based Operating Systems. In other words, polling technique is one of the essential components to implement asynchronous IO.

B. Message-Passing and Synchronization Primitives

Some synchronization primitives can be used as a way to communicate between threads as well as some classical solutions such OpenMP and channels, supported by most part of modern programming languages, including Rust, Kotlin, Dart, TypeScript, Go and others.

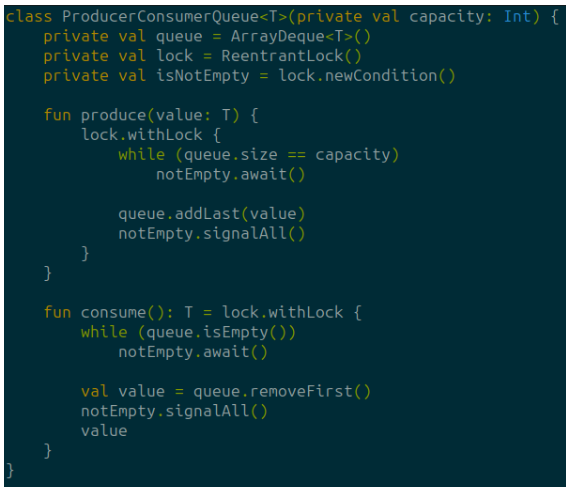
Message-Passing techniques allow to notify different threads about task completion in order to return to execution or provide a callback to handle the result. By working with two sided channel, user will be able to send data from asynchronous worker to current control flowed worker. Fig. 3.3 illustrates message passing technique with channel.

Similar result can be achieved with synchronisation blockers, such as Conditional Variable or Barrier. Fig. 3.4 provides an example of producer-consumer queue with a buffer. When buffer overflows its capacity, queue is waiting for value to remove. If queue is empty, it is also waiting until new element is inserted.



```rust
let barrier = Arc::new(Barrier::new(11));
let pool = ThreadPool::new(10);
let workers = (0..10).map(|n| Arc::new(Worker(n))).collect::<Vec<_>>();
let boss = Boss;

workers
  .iter()
  .for_each(|worker| {
    let barrier = barrier.clone();
    let worker = worker.clone();
    pool.execute(move || worker.start_working(barrier))
  });

boss.come_to_work(barrier.clone());

let (sender, receiver) = channel();

workers
  .into_iter()
  .map(|w| (w, sender.clone()))
  .for_each(|(worker, sender)| {
    pool.execute(move || sender.send(worker.complete_job()).unwrap())
  });

receiver.iter().for_each(|(m, n)| println!("Worker {}: {}", n, m));
pool.join()
```

Fig. 3.3. Communication with Channels between worker threads and master thread



```kotlin
class ProducerConsumerQueue<T>(private val capacity: Int) {
    private val queue = ArrayDeque<T>()
    private val lock = ReentrantLock()
    private val isNotEmpty = lock.newCondition()

    fun produce(value: T) {
        lock.withLock {
            while (queue.size == capacity)
                notEmpty.await()

            queue.addLast(value)
            notEmpty.signalAll()
        }
    }

    fun consume(): T = lock.withLock {
        while (queue.isEmpty())
            notEmpty.await()

        val value = queue.removeFirst()
        notEmpty.signalAll()
        value
    }
}
```

Fig. 3.4. Producer-Consumer queue with conditional variable implementation

III Finite State Automaton and the Event Queue

Notification mechanisms allow to receive signal when event is happen. However, notification is only a signal which cannot execute any callbacks. In other words, notification is sent, but response handling mechanism is still required.

Finite State Automaton (AKA State Machine) is a powerful tool that can easily become a manager which provides asynchronous management of signals and callbacks. Many libraries use FSA under the hood as a runtime to manage both reactive and suspending execution.

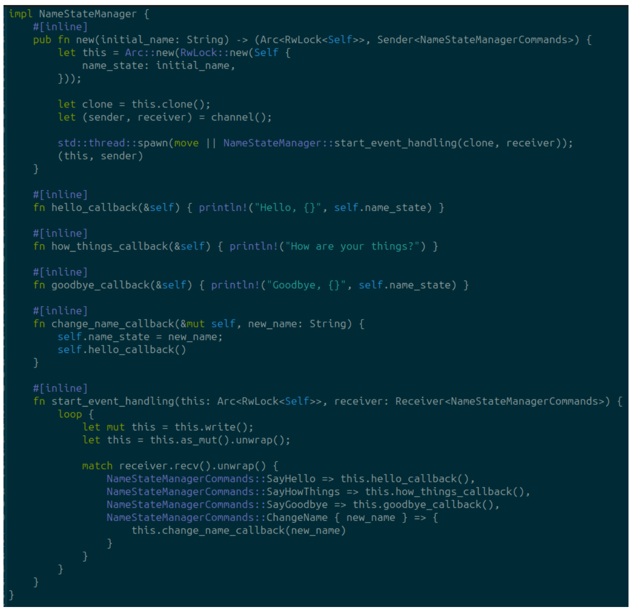
FSA can be described as **a state manager, that controls an order of states' execution, as well as their launch**. It means that this task manager automatically reacts when new state is received. Such FSA manager can use all notification mechanism described above.

Conventionally, any state machine's implementation consists of:

- Set of states, which in our case represent set of events
- State handlers for each state, which in our case are event callbacks
- State transitions, which in our case is the *event queue*

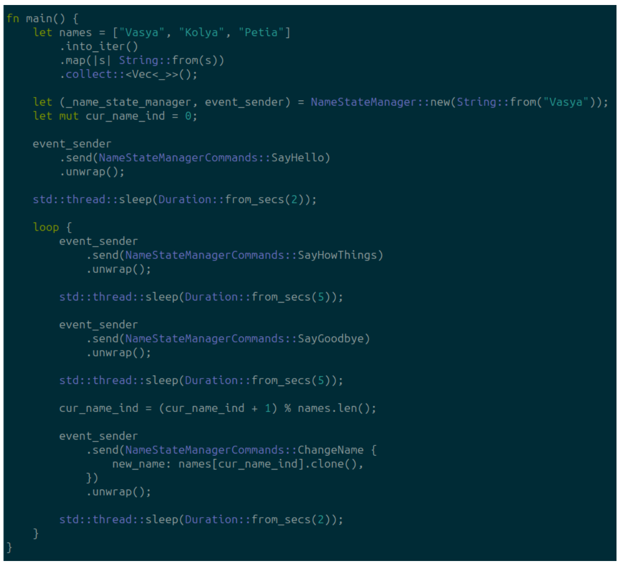
Event Queue can be described as the queue of events/signals, which have to be handled in order as those events arrive to the queue. This queue can be both implicit (e.g. receiving with channels) or explicit (using special data structure). Depending on the environment, this event queue mechanism can be either single threaded (without any thread synchronization techniques) or multithreaded (with thread synchronization techniques like sync primitives).

In other words, event queue's implementation depends on chosen notification mechanism discussed above. And, usually, it wraps one of those mechanisms. Next code sample demonstrates an example of an asynchronous state manager with an event queue implemented as sender and receiver channels (implicit event queue). Fig 3.5-3.6 partially illustrate implementation of the state machine.



```rust
impl NameStateManager {
    #[inline]
    pub fn new(initial_name: String) -> (Arc<RwLock<Self>>, Sender<NameStateManagerCommands>) {
        let this = Arc::new(RwLock::new(Self {
            name_state: initial_name,
        }));

        let clone = this.clone();
        let (sender, receiver) = channel();

        std::thread::spawn(move || NameStateManager::start_event_handling(clone, receiver));
        (this, sender)
    }

    #[inline]
    fn hello_callback(&self) { println!("Hello, {}", self.name_state) }

    #[inline]
    fn how_things_callback(&self) { println!("How are your things?") }

    #[inline]
    fn goodbye_callback(&self) { println!("Goodbye, {}", self.name_state) }

    #[inline]
    fn change_name_callback(&mut self, new_name: String) {
        self.name_state = new_name;
        self.hello_callback()
    }

    #[inline]
    fn start_event_handling(this: Arc<RwLock<Self>>, receiver: Receiver<NameStateManagerCommands>) {
        loop {
            let mut this = this.write();
            let this = this.as_mut().unwrap();

            match receiver.recv().unwrap() {
                NameStateManagerCommands::SayHello => this.hello_callback(),
                NameStateManagerCommands::SayHowThings => this.how_things_callback(),
                NameStateManagerCommands::SayGoodbye => this.goodbye_callback(),
                NameStateManagerCommands::ChangeName { new_name } => {
                    this.change_name_callback(new_name)
                }
            }
        }
    }
}
```

Fig. 3.5. State machine implementation



```rust
fn main() {
    let names = ["Vasya", "Kolya", "Petia"]
        .into_iter()
        .map(|s| String::from(s))
        .collect::<Vec<_>>();

    let (_name_state_manager, event_sender) = NameStateManager::new(String::from("Vasya"));
    let mut cur_name_ind = 0;

    event_sender
        .send(NameStateManagerCommands::SayHello)
        .unwrap();

    std::thread::sleep(Duration::from_secs(2));

    loop {
        event_sender
            .send(NameStateManagerCommands::SayHowThings)
            .unwrap();

        std::thread::sleep(Duration::from_secs(5));

        event_sender
            .send(NameStateManagerCommands::SayGoodbye)
            .unwrap();

        std::thread::sleep(Duration::from_secs(5));

        cur_name_ind = (cur_name_ind + 1) % names.len();

        event_sender
            .send(NameStateManagerCommands::ChangeName {
                new_name: names[cur_name_ind].clone(),
            })
            .unwrap();

        std::thread::sleep(Duration::from_secs(2));
    }
}
```

Fig. 3.6. Pushing events to the event queue

A. Android Event Loop: Handler, Looper and MessageQueue

One of the biggest example of asyncrhonous UI implementation with an explicit Event Queue implementation and usage is Android SDK. Android Framework is one of the biggest State Machine implementations that exists nowadays. UI management is implemented together with its own event queue, handled by three core Android components: *Handler*, *Looper* and *MessageQueue with Messages*.

Message is an object which is **passed between event handlers**. Android's message represents and event, which is either an information (which can be passed either as an integer flag, or as a Parcelable object). Study by Vasily Drobushkov [8] describes Messages in more details. Fig. 3.7 illustrates Messages. First one passes custom parcelizable object, second one passes Runnable callback.

```
@Serializable
data class VideoMetadata(
    val title: String,
    val author: String,
    val covers: List<String>,
    val lenInMillis: Long
) : Parcelable

val handler = Handler(Looper.getMainLooper())
val id = 1
val obj = VideoMetadata()

// Message with a custom object
Message.obtain(handler, id, obj)

// Message with an action
Message.obtain(handler) { Log.d(TAG, "Hello from MainActivity") }
```

Fig. 3.7. Example of custom Messages

MessageQueue is what you think is is: linked list implemented buffer that stores events that are processed afterwards. In other words, it is a part of local event queue mechanism: it **helps to store and acquire events to send them to the event handlers, when it is required**. Since Android Framework is based on JVM, which conceptually since the beginning provides thread API, MessageQueue uses monitors (synchronized blocks) to provide *synchronization when multiple receivers try to acquire data*. For more details, take a look at the source code.

Looper is an object that **runs message loop inside a single thread**. Basically, it is responsible for **putting Messages from the MessageQueue and accessing them**. Each thread used and managed by the Android Framework is fully occupied with a single Looper and its own MessageQueue. Fig. 3.8 illustrates a sample thread that is prepared for a Looper's management.

Handler is an object that is responsible for **sending and receiving Messages from the Looper**. Each Handler is **assigned to the single Looper and its single thread with MessageQueue**. Basically, Handler is the mediator between UI layer and

```
val thread = thread {
    Looper.prepare()
    Looper.loop()
    Log.d(TAG, "Hello from the Looper ${Looper.myLooper()}")

    Looper.myQueue().addIdleHandler {
        Log.d(TAG, "Message queue has finished all its tasks")
    }
}
```

Fig. 3.8. Custom thread prepared for a Looper

the MessageQueue, which is an Event Queue for every Android application. It sends data and callbacks processed and executed on the Looper's thread. Mainly, Handler provides message-passing functionality: it sends events from the current thread to threads identified with a Looper's instance. Fig. 3.9 demonstrates a background thread that sends messages through a Handler assigned to the Main Looper.

```
thread {
    val handler = object : Handler(Looper.getMainLooper()) {
        override fun handleMessage(msg: Message) {
            super.handleMessage(msg)
            Log.d(TAG, "Callback is received on the main thread")
            msg.callback.run()
        }
    }

    handler.sendMessage(
        Message.obtain(handler) {
            Log.d(TAG, "Hello from MainActivity")
        }
    )
}
```

Fig. 3.9. Background thread sends messages to the main thread

Event Loop in Android Framework is a quite extensive approach to implement asynchronous execution for UI management. It is indeed complicated and not as straightforward as in modern solutions, that is why our study was not focused on the details of implementation, but rather conceptions that is rely on. Principles that are implemented are still powerful and relevant for all modern applications.

In conclusion, Android Event Loop consists of s**pecial threads mainly assigned to the event management**. Together with Handler components **events can be send and acquired**, which provides opportunity to manage tasks asynchronously and communicating with different threads. Fig 3.10 provides a scheme of Android's Event Loop.

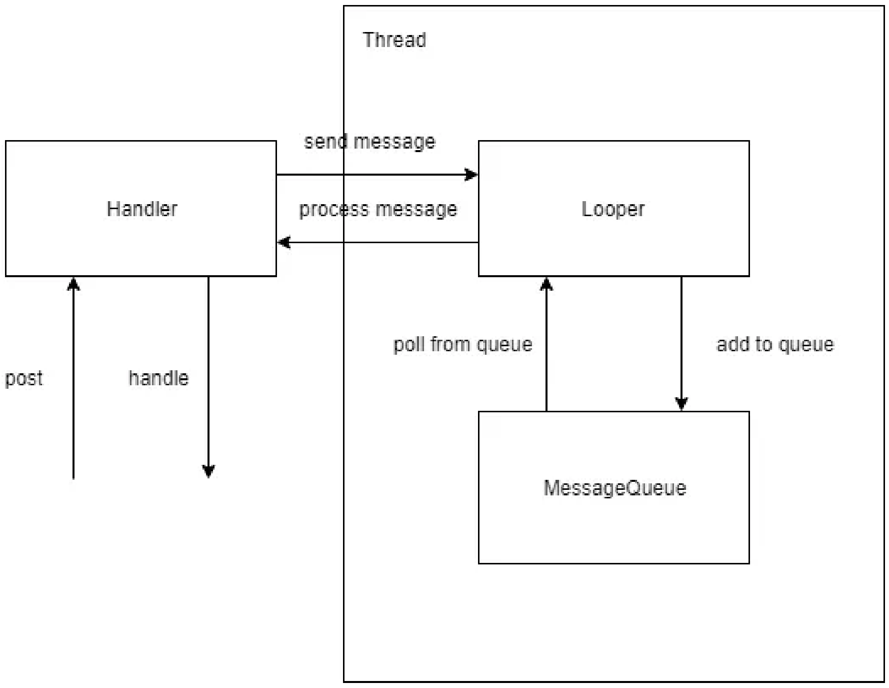


Fig. 3.10. Event Loop in Android with Handler, Looper and MessageQueue

IV Going deep into Reactive Streams

Reactive programming can be considered as streams of data, which are observed by a handler. According to Clement Escoffier's article [9], all observers that subscribe to updates receive notifications about updates and react to such updates in asynchronous way. It means that developer and system usually cannot predict exact time when this notification arrives (in other words, it is hard to use acquired value in main control flow). However, when it arrives, some callback handles this update.

Usually, there are two types of such streams: Producers and Mutators.

Producers are streams of data that update their value states by **adding the new one to the collection** of data. In other words, if something is changed, producer adds new data to the event queue. After that, new value will be received by the observers that had subscribed to producer.

Under the hood, this workflow can be achieved with the ***"Factory"*** pattern, described by Gamma et al. [6]. More preciseful explanation: callback is introduced in order to construct new data, depending on the previous data (which is optional) and some other data, captured by the callback. Fig. 3.11 provides UML diagram for Factory pattern.

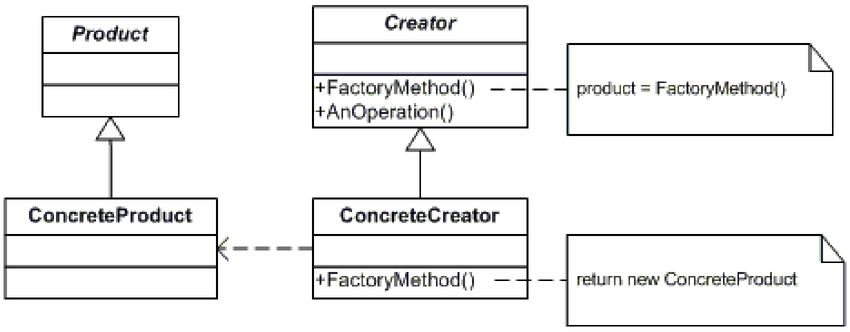
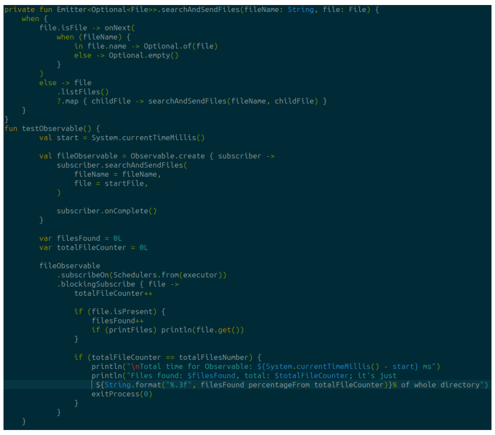


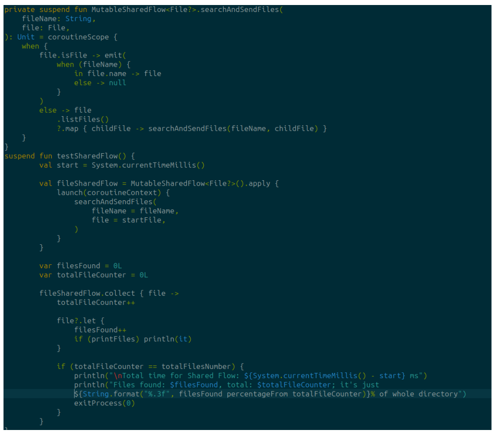
Fig. 3.11. UML diagram of Factory pattern

As an example, SharedFlow from Kotlin's Coroutines library share this idea. Observables from RxJava2, such as Observable and Flowable. For Rust language, RxRust crate provide similar functionality to RxJava2. Fig. 3.12-3.14 illustrate producers discussed above in action.



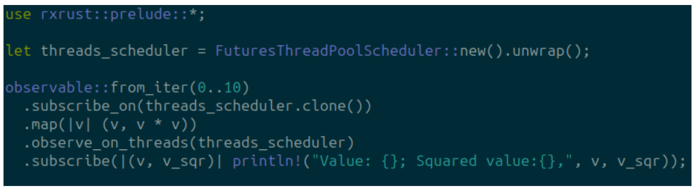
```
private fun Emitter<Optional<File>>.searchAndSendFiles(fileName: String, file: File) {
    when {
        file.isFile -> onNext(
            when (fileName) {
                in file.name -> Optional.of(file)
                else -> Optional.empty()
            }
        )
        else -> file
            .listFiles()
            ?.map { childFile -> searchAndSendFiles(fileName, childFile) }
    }
}
fun testObservable() {
        val start = System.currentTimeMillis()

        val fileObservable = Observable.create { subscriber ->
            subscriber.searchAndSendFiles(
                fileName = fileName,
                file = startFile,
            )

            subscriber.onComplete()
        }

        var filesFound = 0L
        var totalFileCounter = 0L

        fileObservable
            .subscribeOn(Schedulers.from(executor))
            .blockingSubscribe { file ->
                totalFileCounter++

                if (file.isPresent) {
                    filesFound++
                    if (printFiles) println(file.get())
                }

                if (totalFileCounter == totalFilesNumber) {
                    println("\nTotal time for Observable: ${System.currentTimeMillis() - start} ms")
                    println("Files found: $filesFound, total: $totalFileCounter; it's just
                     ${String.format("%.3f", filesFound percentageFrom totalFileCounter)}% of whole directory")
                    exitProcess(0)
                }
            }
    }
```

Fig. 3.12. Counting files with same name using Observable



```
private suspend fun MutableSharedFlow<File?>.searchAndSendFiles(
    fileName: String,
    file: File,
): Unit = coroutineScope {
    when {
        file.isFile -> emit(
            when (fileName) {
                in file.name -> file
                else -> null
            }
        )
        else -> file
            .listFiles()
            ?.map { childFile -> searchAndSendFiles(fileName, childFile) }
    }
}
suspend fun testSharedFlow() {
        val start = System.currentTimeMillis()

        val fileSharedFlow = MutableSharedFlow<File?>().apply {
            launch(coroutineContext) {
                searchAndSendFiles(
                    fileName = fileName,
                    file = startFile,
                )
            }
        }

        var filesFound = 0L
        var totalFileCounter = 0L

        fileSharedFlow.collect { file ->
            totalFileCounter++

            file?.let {
                filesFound++
                if (printFiles) println(it)
            }

            if (totalFileCounter == totalFilesNumber) {
                println("\nTotal time for Shared Flow: ${System.currentTimeMillis() - start} ms")
                println("Files found: $filesFound, total: $totalFileCounter; it's just
                ${String.format("%.3f", filesFound percentageFrom totalFileCounter)}% of whole directory")
                exitProcess(0)
            }
        }
```

Fig. 3.13. Counting files with same name using SharedFlow



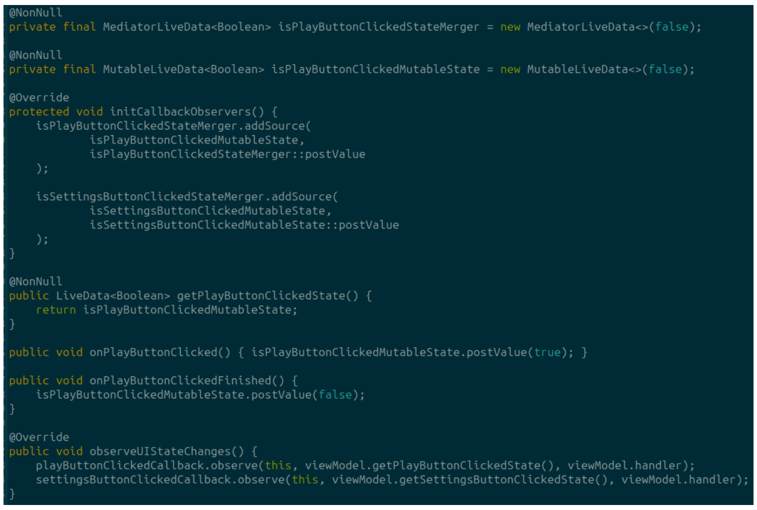
```
use rxrust::prelude::*;

let threads_scheduler = FuturesThreadPoolScheduler::new().unwrap();

observable::from_iter(0..10)
  .subscribe_on(threads_scheduler.clone())
  .map(|v| (v, v * v))
  .observe_on_threads(threads_scheduler)
  .subscribe(|(v, v_sqr)| println!("Value: {}; Squared value:{},", v, v_sqr));
```

Fig. 3.14. Computing the square of the number with RxRust

Mutators are those data streams that usually **update their value**, rather than use any buffering technique. Since the value can be updated concurrently, **thread safety**

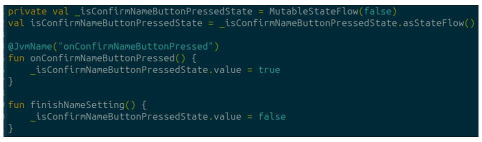
method is introduced, such as ***synchronization*** with mutex-like synchronization primitive or ***atomic computations***. Usually, second approach is preferred, since atomic calculations are more lightweight than thread synchronization.

As an example, StateFlow from Kotlin's Coroutines, LiveData from Android Framework and Hooks from ReactJS share idea of mutability. Fig. 3.15-3.17 illustrate mutators.



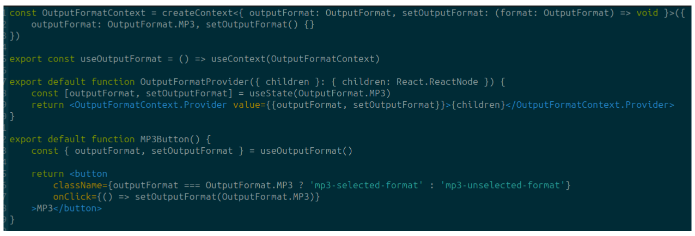
```java
@NonNull
private final MediatorLiveData<Boolean> isPlayButtonClickedStateMerger = new MediatorLiveData<>(false);

@NonNull
private final MutableLiveData<Boolean> isPlayButtonClickedMutableState = new MutableLiveData<>(false);

@Override
protected void initCallbackObservers() {
    isPlayButtonClickedStateMerger.addSource(
            isPlayButtonClickedMutableState,
            isPlayButtonClickedStateMerger::postValue
    );

    isSettingsButtonClickedStateMerger.addSource(
            isSettingsButtonClickedMutableState,
            isSettingsButtonClickedMutableState::postValue
    );
}

@NonNull
public LiveData<Boolean> getPlayButtonClickedState() {
    return isPlayButtonClickedMutableState;
}

public void onPlayButtonClicked() { isPlayButtonClickedMutableState.postValue(true); }

public void onPlayButtonClickedFinished() {
    isPlayButtonClickedMutableState.postValue(false);
}

@Override
public void observeUIStateChanges() {
    playButtonClickedCallback.observe(this, viewModel.getPlayButtonClickedState(), viewModel.handler);
    settingsButtonClickedCallback.observe(this, viewModel.getSettingsButtonClickedState(), viewModel.handler);
}
```

Fig. 3.15. LiveData (Java) observes state updates



```kotlin
private val _isConfirmNameButtonPressedState = MutableStateFlow(false)
val isConfirmNameButtonPressedState = _isConfirmNameButtonPressedState.asStateFlow()

@JvmName("onConfirmNameButtonPressed")
fun onConfirmNameButtonPressed() {
    _isConfirmNameButtonPressedState.value = true
}

fun finishNameSetting() {
    _isConfirmNameButtonPressedState.value = false
}
```

Fig. 3.16. StateFlow (Kotlin) observes state updates



```tsx
const OutputFormatContext = createContext<{ outputFormat: OutputFormat, setOutputFormat: (format: OutputFormat) => void }>({
    outputFormat: OutputFormat.MP3, setOutputFormat() {}
})

export const useOutputFormat = () => useContext(OutputFormatContext)

export default function OutputFormatProvider({ children }: { children: React.ReactNode }) {
    const [outputFormat, setOutputFormat] = useState(OutputFormat.MP3)
    return <OutputFormatContext.Provider value={{outputFormat, setOutputFormat}}>{children}</OutputFormatContext.Provider>
}

export default function MP3Button() {
    const { outputFormat, setOutputFormat } = useOutputFormat()

    return <button
        className={outputFormat === OutputFormat.MP3 ? 'mp3-selected-format' : 'mp3-unselected-format'}
        onClick={() => setOutputFormat(OutputFormat.MP3)}
    >MP3</button>
}
```

Fig. 3.17. Custom ReactJS Hook implementation and usage

Producers and Mutators differ in idea of updating values. ***Producers*** implement the idea of ***Message-Passing interface*** by storing value to the **data queue**, which may have different buffering strategy (e.g. limited and unlimited buffering, removing old values if buffering capacity is reached, throwing error and etc.). ***Mutators*** concurrently (if supported, for example, Flow from Kotlin allows only single threaded management) **update their value and notify observers** with OS-based polling techniques discussed above, with synchronization primitives or with suspending execution.

V Cold and Hot Streams

As it was mentioned, Reactive programming is based on asynchronous data streams that manage data. Such streams are differ not only in buffering strategy, but in ways they produce notifications to their observers. These data streams are distributed into two different types: *"Cold"* and *"Hot"* streams, which have next unique strategies:

- Data production strategy (inside or outside of producer)
- Message-receiving strategy (single or broadcast)
- Data sharing strategy (lazily, while subscribed, eagerly)

A. Cold Streams

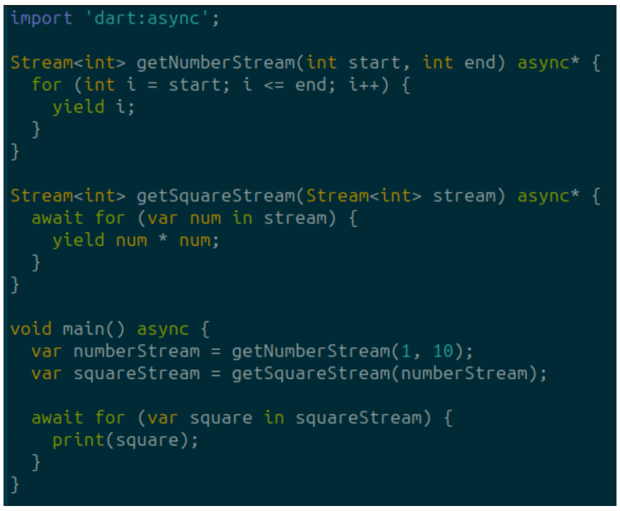
Cold Stream can be described as a **lazy inside-producers with only single observer**. According to Julián Ezequiel Picó [10], this single observer is *the one, who initializes the whole stream.* In other words, **every new subscriber creates its own instance of a cold stream**, which will produce new values independently from other instances.

According to Roman Elizarov [11], Cold Streams stop producing value when it is not required by the listener. Since such producers have only one receiver, no synchronization is required, because data can be updated only in single thread (even if not the same thread on each update).

As an example, Flow from Kotlin, Stream from Rust (extensions for Tokio crate) and Streams from Dart can be described as cold flows, which lazily produce values and have single receiver. Fig. 3.18-3.19 show Dart and Rust streams in action.

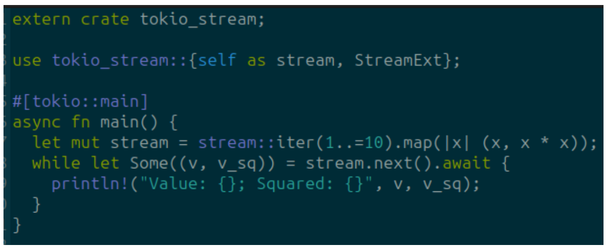
B. Hot Streams

Hot Stream can be described as an **eager or while-subscribed outside-producers with multiple or zero observers**. According to Julián Ezequiel Picó [10], hot streams use **broadcasting idea**: they send data to some unknown receivers. Besides, hot streams do not know if someone is listening to them - they only produce new values. In



```dart
import 'dart:async';

Stream<int> getNumberStream(int start, int end) async* {
  for (int i = start; i <= end; i++) {
    yield i;
  }
}

Stream<int> getSquareStream(Stream<int> stream) async* {
  await for (var num in stream) {
    yield num * num;
  }
}

void main() async {
  var numberStream = getNumberStream(1, 10);
  var squareStream = getSquareStream(numberStream);

  await for (var square in squareStream) {
    print(square);
  }
}
```

Fig. 3.18. Dart's Stream producing squared values



```rust
extern crate tokio_stream;

use tokio_stream::{self as stream, StreamExt};

#[tokio::main]
async fn main() {
  let mut stream = stream::iter(1..=10).map(|x| (x, x * x));
  while let Some((v, v_sq)) = stream.next().await {
    println!("Value: {}; Squared: {}", v, v_sq);
  }
}
```

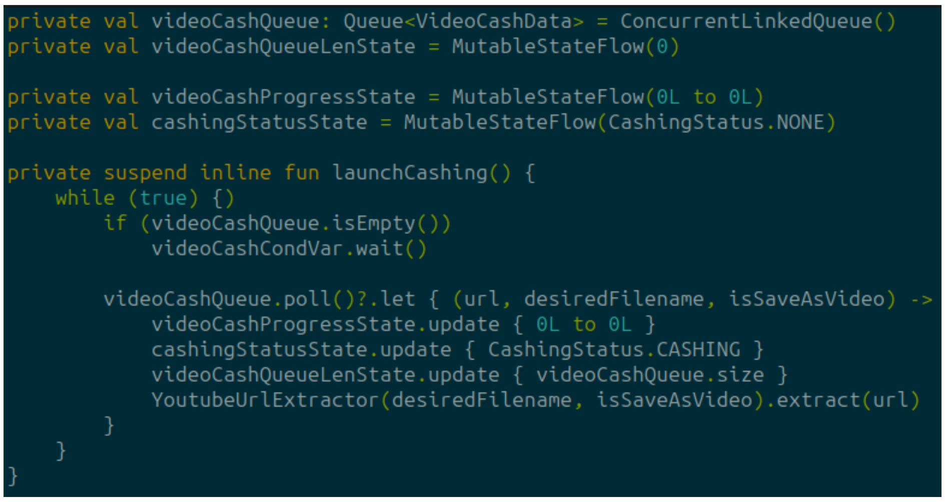
Fig. 3.19. Rust's Tokio Stream producing squared values

simple words, it can be described as radio station: someone is talking and changing music, while others listen to this person's speech or dance to the rhythm. Or maybe radio host is just talking to himself and nobody listen - it does not matter.

According to Roman Elizarov [11], Hot Streams are **independent from application's requests**: during production they may depend on additional data as well as other hot and cold streams, however, they are **not allowed to require any bonds to listeners**. Besides, hot streams **require synchronization**: producers may concurrently update and send their values, so they require either synchronization primitives or atomic operations (e.g. StateFlow is implemented as atomic mutator). Fig. 16 demonstrates atomic interface of StateFlow, which is very similar to the way atomic variables, such as AtomicInteger works in JVM languages.

As an example, StateFlow from Kotlin's Coroutines, Observable from RxJava2 and ObservableBase from RxRust share the idea of Hot Streams.

Besides, modern UI libraries, based on React, such as React itself, Compose, Flutter, Yew for WebAssembly Frontend development with Rust and others support the idea of "recomposition" - property to rerun UI components in asynchronous way if some changes to UI data are made.



```
private val videoCashQueue: Queue<VideoCashData> = ConcurrentLinkedQueue()
private val videoCashQueueLenState = MutableStateFlow(0)

private val videoCashProgressState = MutableStateFlow(0L to 0L)
private val cashingStatusState = MutableStateFlow(CashingStatus.NONE)

private suspend inline fun launchCashing() {
    while (true) {
        if (videoCashQueue.isEmpty())
            videoCashCondVar.wait()

        videoCashQueue.poll()?.let { (url, desiredFilename, isSaveAsVideo) ->
            videoCashProgressState.update { 0L to 0L }
            cashingStatusState.update { CashingStatus.CASHING }
            videoCashQueueLenState.update { videoCashQueue.size }
            YoutubeUrlExtractor(desiredFilename, isSaveAsVideo).extract(url)
        }
    }
}
```

Fig. 3.20. Multiple StateFlows atomically updates values after cashing event

VI Last words about Rx

Reactive execution consists of an **event queue** mechanism. It may be explicit or implicit. Events can be sent and received. Usually, **objects implemented as the "Observer" pattern** are used to handle notifications (events) in special callbacks asynchronously. Event/notification observers are allowed to **communicate with multiple threads** with thread safety mechanism and a **special scheduler "thread pool"**, or they can manage requests **within a single thread**.

Chapter 4

Suspending execution

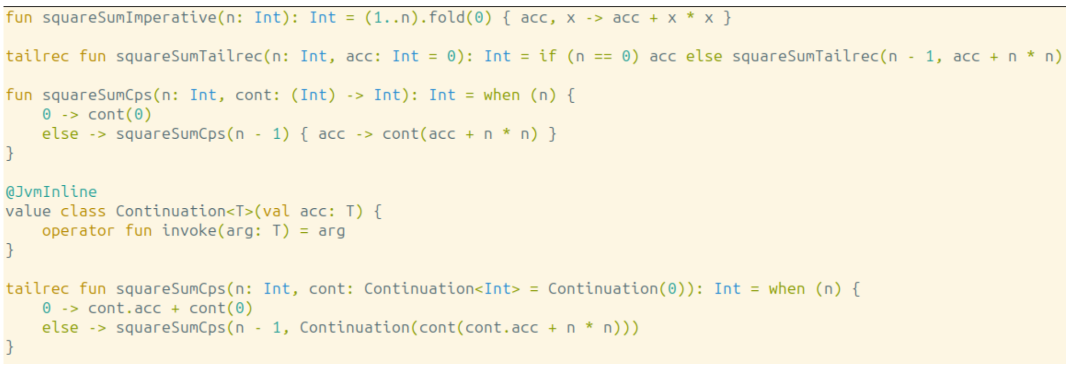
Modern solution for an asynchronous workflow, async/await feature makes it possible to continue current control flow as it was typical sequential execution. In the same time it allows to work with asynchronous operations (such as IO operations) and use their results without additional callbacks, if they are not required. Implement async/await feature relies on the conception of *suspending execution*.

I Idea of suspending execution

Suspending execution is the execution that can dynamically switch its control flow when it is necessary. Usually, suspending execution is marked with a special keyword, such as *async* keyword, or, for Kotlin it is *suspend* keyword. Those keywords mark points in control flow, where the control flow can be switched to other tasks while current function is suspended. In other words, such suspend functions allows to suspend their execution and resume when it is appropriate (for example, after receiving data from server or launching SQL SELECT query). Once the suspending block of code is completed, the function can resume its execution from where it left off. This approach keeps control flow as it was with sequential execution or threading, meaning that developer can explicitly use acquired value from suspended code without any message-passing techniques, callbacks or control flow breaks.

II Continuation-Passing Style

Most part of main-stream libraries use implementation of suspending control flow that relies on ***"Continuation-Passing style"***. According to the next article about CPS and its role in Functional Paradigm [12], technique is almost identical to tail recursion, however, CPS implementation passes not accumulated value, but the function instead, meaning that **all CPS-based functions are high-order functions**. Fig. 4.1 demonstrates differences between imperative, tail recursive and CPS styles.



```kotlin
fun squareSumImperative(n: Int): Int = (1..n).fold(0) { acc, x -> acc + x * x }

tailrec fun squareSumTailrec(n: Int, acc: Int = 0): Int = if (n == 0) acc else squareSumTailrec(n - 1, acc + n * n)

fun squareSumCps(n: Int, cont: (Int) -> Int): Int = when (n) {
    0 -> cont(0)
    else -> squareSumCps(n - 1) { acc -> cont(acc + n * n) }
}

@JvmInline
value class Continuation<T>(val acc: T) {
    operator fun invoke(arg: T) = arg
}

tailrec fun squareSumCps(n: Int, cont: Continuation<Int> = Continuation(0)): Int = when (n) {
    0 -> cont.acc + cont(0)
    else -> squareSumCps(n - 1, Continuation(cont(cont.acc + n * n)))
}
```

Fig. 4.1. Sums of squares in imperative, tail rec and CPS styles

Continuation-Passing style allows to separate block of code on parts between async/await points. According to Simon Vergauwen's research [13], when control flow reaches any of such points, **continuation function is stored into coroutine's stack**. Then control flow **switches to one of the continuations that are ready to execute**. When the async operation is finished, **coroutine returns to the saved continuation function and continues its execution** as it was in the sequential code. Fig. 4.2 illustrates a scheme of how tasks are managed in suspending execution.

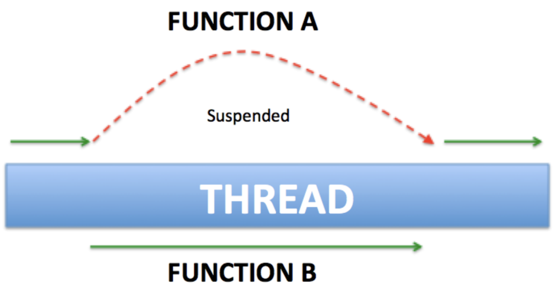


Fig. 4.2. Suspending execution

A. Kotlin's style for suspending CPS execution

Suspend functions in Kotlin is one of the most powerful implementation of such concept, yet, it is mainly based on CPS with additional features and upgrades. For this section I have prepared a special experimental example of both Kotlin's suspend function with multiple suspend points and its decompiled Java byte-code. Second one is too massive to show as a picture, so I expect you to follow the links above.

Experimental environment: For this experiment I have chosen a function that switches multiple times from synchronous tasks to asynchronous ones. Function itself prints "hello" message multiple times. In order to separate suspending points, different delay ranges between messages were chosen.

Decompiled output was generated with IntelliJ's feature "Show Kotlin Bytecode" with all compiler's optimizations enabled.

Algorithm to suspend function: As it can be seen, function is separated on two main blocks (label47 and label40): *initializer* and *manager*.

Initializer is the block that initializes *next continuation*. Here, continuation is an object that will call the same function again, when it is required. Basically, Kotlin's continuation is the mechanism that helps to return to the function in order to continue execution after suspend point was reached and given suspend function was executed.

Structure of continuation consists of next features:

- ***label***, which is the current position in function in terms of suspending points. It can be described as the number of reached suspend points on the current function's execution. Label with value greater than the number of suspend points indicates that executional error was happen.
- ***result*** of the function, which is typed as basic Result of Kotlin
- ***Current CoroutineScope*** that helps to manage both current and other coroutines.

Even if it cannot be seen explicitly, continuation is definitely stored in the queue. Later, it is removed and passed to its own invoke() call.

As was tested on multiple "hello()" function variations, this block will always looks the same, even in other suspend functions.

Manager, on the other side, is always unique, yet, has a generation algorithm, which depends on number of suspend points in the function.

For each new suspend point special block is created. All such blocks are nested in *LIFO (stack) order* (executional block after the last suspend point will be the outermost). All those blocks look the same except the innermost.

Innermost block contains switch/cases statement that helps to manage blocks according to the current label (number of suspending points reached). As it can be seen, label "0" corresponds to the first executional block, when first suspend point is reached. For other labels, it jumps to the required block according to the label.

Function becomes suspended in the innermost block with IntrinsicsKt.getCOROUTINE-SUSPENDED(), so in this moment control flow have to switch to function within current thread. Function is suspended until *the suspend block is finished and control flow is returned back to this function. For each getCOROUTINE-SUSPENDED() call new Continuation is generated in the* "initializer" *block.*

Since function executes almost recursively, it can be trivially seen that function executes in *"onion"* way: on each coroutine's suspending call with IntrinsicsKt.getCOROUTINE-SUSPENDED(), *control flow travels to the innermost block, then with switch/case it jumps to the outer block of code according to the current function execution's number*, which is updated within Continuation in each block in executional section (as well as the current coroutine scope). So, on the first call we travel only to the innermost block, then we jump to the second innermost block and so on. On the last step control flow still travels to the innermost block, the jumps to the outermost.

As it can be noticed, *each executional block returns a result of the current block, which is probably represents the stack memory*, cashed on the current suspend step. In other words, *each function's call returns all dependencies that will be restored* when function is triggered again by the Continuation.

Idea itself is quite powerful and extensible, however, this mechanism cannot be implemented on the pure Java and requires serious *code generation mechanism*, as well as magic of internal compiler's behaviour.

III What is Coroutine?

Basically, **coroutine is a lightweight thread that supported by the compiler itself**, not by Operating System. As a thread, **it has its own stack with all variables that are local to its continuation**.

The main problem of threads are data race and race conditions. According to the article by Karthik [14], *data race is an event when multiple control flows tries to access same memory. Race conditions are conditions when program tries to execute multiple commands on the same memory.* Since coroutines do not have their own control flow and tries to manipulate source control flow, **race conditions cannot happen, because coroutines will manage their assigned control flow one after another, but not in the same time**. Data races can happen only in the case **when coroutines were created in different threads**. In this case developer has to use special synchronization primitives to **synchronize coroutines from different threads, but not threads themselves**, since all other coroutines that were launched from synchronized threads will be blocked because of control flow blocking.

Coroutines allow to avoid thread blocks, caused by long-waiting operations by switching control flow to the next continuation. However, in case of multithreaded execution, they require special synchronization primitives, that conceptually based on same sync primitives that are used in multithreaded environment (such as mutexes, semaphores, condition variables, read-write locks, barriers and etc.).

Coroutines are not threads, meaning that they lack their own control flow. In other words, it is not beneficial to launch high-computational task that does not relies on coroutines' suspense. This operation will require all its thread's computational power, meaning that it will block the thread.

IV Real cases of Coroutines

Nowadays multiple modern languages use coroutines and suspending execution as the way to implement asynchronous execution. This article will focus on Kotlin, Rust and Go because of their modern and clean from conventional point of view approach to the implementation of coroutines and suspending execution. All 3 implementations share concepts, yet, they introduce different approaches for the implementation of the same features.

A. Kotlin

Kotlin's Coroutines are the mainstream feature and one of the reasons why many developers switched to Kotlin. Coroutines library provides extremely convenient way to manage coroutines efficiently.

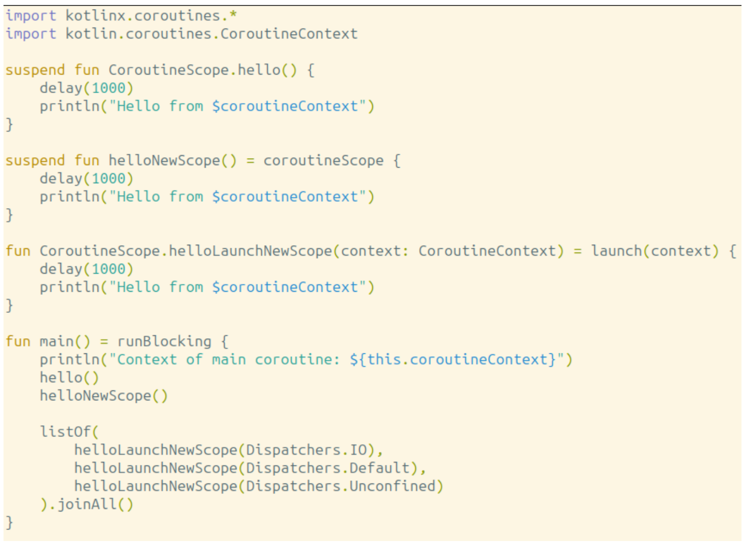
Coroutine's creation: Coroutines have their own Scope that allows to execute suspending functions. Common way to create main app's coroutine is to run main() function with runBlocking function, as shown in Fig. 4.3. This builder blocks the thread until all Continuations that are created in this block have finished their tasks. From the created Coroutine Scope new Coroutines can be created with async and launch extension functions, depending on purpose of execution: *async()* allows to return value from Coroutine, *launch()* executes block of code without any returning values.

Dispatchers: Internally, Kotlin uses ***thread pool*** to manage different Coroutines according to their purpose. According to Amit Shekhar's article [15], dispatching of Coroutines is distributed in the next way:

- Default dispatcher is designed for high-computational and CPU-intensive tasks, e.g. computation of matrices' determinant
- IO dispatcher is designed for IO operations for both file system and network requests, e.g. database query, file management or GET request from remote API via HTTP client
- Main is designed for UI updates only. It means that this dispatcher is single-threaded and requires UI layer in the app (for example, UI layer in Android and UI thread). Note that UI thread is not always main thread, meaning that UI thread and main function's thread can be unequal to each other
- Unconfined uses same thread that has created the Coroutine. In other words, this dispatcher is useful when context of the execution does not matter or we explicitly

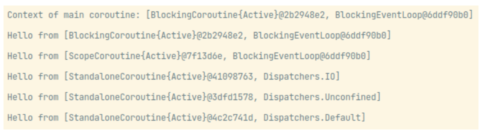
want to work on the same thread (for instance, to manage tasks on Main thread (not UI))

Fig. 4.3-4.4 illustrates an example of execution of the code with multiple coroutines from different dispatchers.



```kotlin
import kotlinx.coroutines.*
import kotlin.coroutines.CoroutineContext

suspend fun CoroutineScope.hello() {
    delay(1000)
    println("Hello from $coroutineContext")
}

suspend fun helloNewScope() = coroutineScope {
    delay(1000)
    println("Hello from $coroutineContext")
}

fun CoroutineScope.helloLaunchNewScope(context: CoroutineContext) = launch(context) {
    delay(1000)
    println("Hello from $coroutineContext")
}

fun main() = runBlocking {
    println("Context of main coroutine: ${this.coroutineContext}")
    hello()
    helloNewScope()

    listOf(
        helloLaunchNewScope(Dispatchers.IO),
        helloLaunchNewScope(Dispatchers.Default),
        helloLaunchNewScope(Dispatchers.Unconfined)
    ).joinAll()
}
```

Fig. 4.3. Launching hello from multiple coroutines with different dispatchers



```
Context of main coroutine: [BlockingCoroutine{Active}@2b2948e2, BlockingEventLoop@6ddf90b0]
Hello from [BlockingCoroutine{Active}@2b2948e2, BlockingEventLoop@6ddf90b0]
Hello from [ScopeCoroutine{Active}@7f13d6e, BlockingEventLoop@6ddf90b0]
Hello from [StandaloneCoroutine{Active}@41098763, Dispatchers.IO]
Hello from [StandaloneCoroutine{Active}@3dfd1578, Dispatchers.Unconfined]
Hello from [StandaloneCoroutine{Active}@4c2c741d, Dispatchers.Default]
```

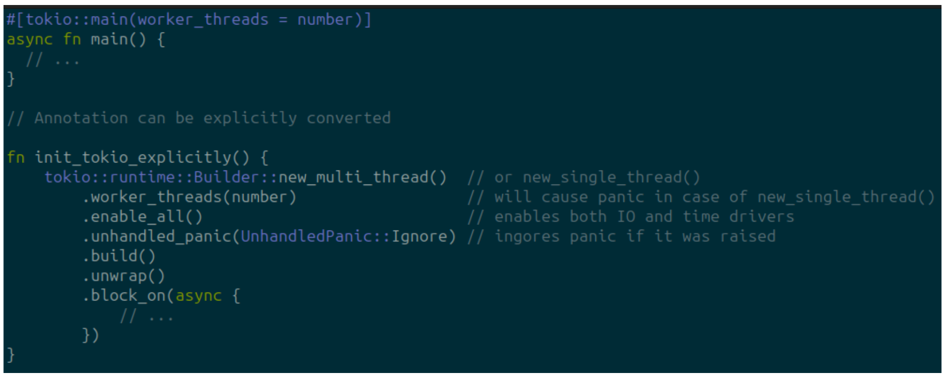
Fig. 4.4. Output of the simulation

B. Rust

Rust's standard library contains only the bindings to Futures's type and async/await keywords. It has not got official library for asynchronous workflow, however, community (which is also consists of language's developers) has created some crates that implement asynchronous workflow in similar way to Kotlin's Coroutines. Nowadays, Tokio crate is used most often as it is already has stable release and provides powerful and deep functionality with a lot of features.

Tokio Runtime: Tokio provides its own runtime to handle asynchronous workflow. Under the hood, Tokio's runtime is a thread pool, that manages Coroutine's distribution. However, it does not have such distribution logic as in Kotlin.

Runtime itself is the core of Tokio, it manages threads, error handling, additional modules and coroutines themselves. Fig. 4.5 shows how this runtime can be created and configured.



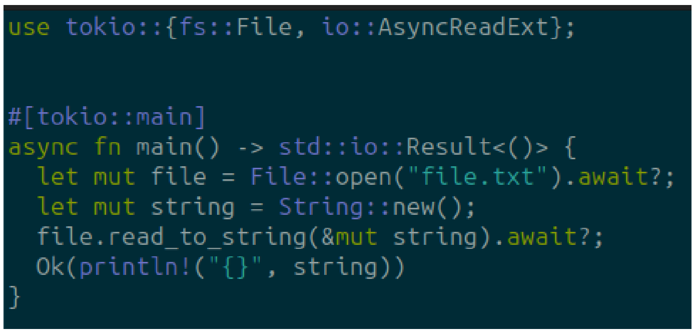
```
#[tokio::main(worker_threads = number)]
async fn main() {
  // ...
}

// Annotation can be explicitly converted

fn init_tokio_explicitly() {
    tokio::runtime::Builder::new_multi_thread()  // or new_single_thread()
        .worker_threads(number)                  // will cause panic in case of new_single_thread()
        .enable_all()                            // enables both IO and time drivers
        .unhandled_panic(UnhandledPanic::Ignore) // ingores panic if it was raised
        .build()
        .unwrap()
        .block_on(async {
            // ...
        })
}
```

Fig. 4.5. Initialization of Tokio runtime

Coroutine's creation: Coroutines in Tokio can be created explicitly with the created environment from main function, or with extension methods from tokio::task module. Three main methods are used:

- spawn, which launches Coroutine with non-blocking task in some thread, that do not have to be blocked with CPU-intensive task
- spawn-local that launches Coroutine in the same thread
- spawn-blocking that is designed for CPU-intensive tasks, that may block the thread

File system extensions: Tokio provides asynchronous API for file management. Module tokio::fs contains wrapper over standard File, but with asynchronous API. Fig. 4.6 demonstrates Tokio's file wrapper.

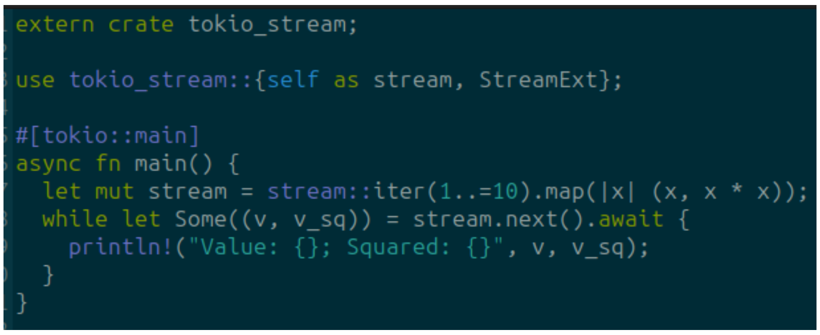


```
use tokio::{fs::File, io::AsyncReadExt};


#[tokio::main]
async fn main() -> std::io::Result<()> {
  let mut file = File::open("file.txt").await?;
  let mut string = String::new();
  file.read_to_string(&mut string).await?;
  Ok(println!("{}", string))
}
```

Fig. 4.6. File management with Tokio File's wrapper

Since file management is OS dependent (Unix-based OS provides asynchronous API with polling, Windows has only blocking interface), it is recommended to run file IO tasks with spawn-blocking() method to improve the performance.

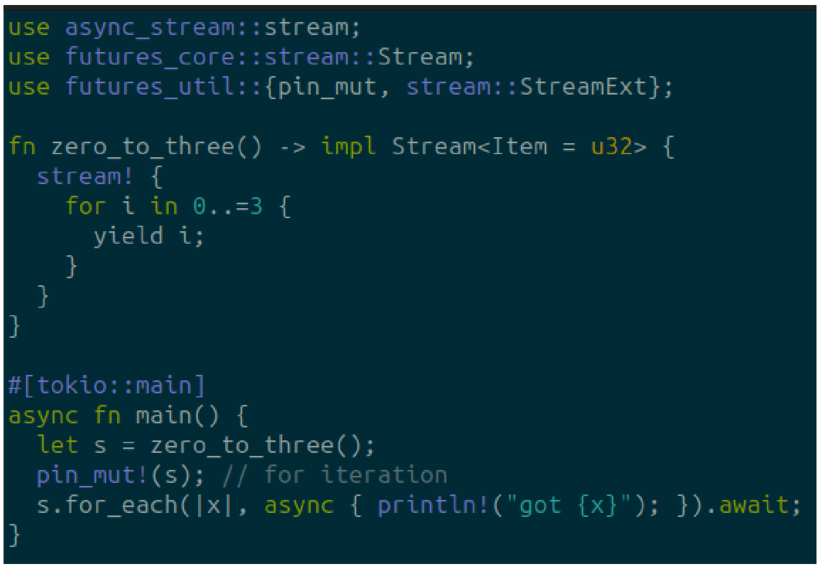
Streams: Tokio provides API for Streams, which are asynchronous functional units (such as iterators in Rust). Basically, Stream is just an iterator over asynchronous tasks with the same return type (polymorphically). Crate futures-rs provides extensions to Stream API with standard functional methods, such as transformators, filters and combiners. Fig. 4.7 illustrates Stream API in action.



```rust
extern crate tokio_stream;

use tokio_stream::{self as stream, StreamExt};

#[tokio::main]
async fn main() {
  let mut stream = stream::iter(1..=10).map(|x| (x, x * x));
  while let Some((v, v_sq)) = stream.next().await {
    println!("Value: {}; Squared: {}", v, v_sq);
  }
}
```

Fig. 4.7. Tokio Stream API in action

Stream in Tokio is a "Cold Stream", meaning that they have to be explicitly collected in order to start producing values. Besides, Streams can insert values only within their body, as Flows in Kotlin. With keyword *yield* values can be added to the current stream. Fig. 4.8 shows Cold Stream API for Tokio Streams.



```rust
use async_stream::stream;
use futures_core::stream::Stream;
use futures_util::{pin_mut, stream::StreamExt};

fn zero_to_three() -> impl Stream<Item = u32> {
  stream! {
    for i in 0..=3 {
      yield i;
    }
  }
}

#[tokio::main]
async fn main() {
  let s = zero_to_three();
  pin_mut!(s); // for iteration
  s.for_each(|x|, async { println!("got {x}"); }).await;
}
```

Fig. 4.8. Tokio Stream can produce values with keyword 'yield'

Synchronization: Concurrency issues can be solved with local synchronization primitives, designed exactly for this purpose. Instead of thread blocking, those sync primitives block only the current coroutine. Tokio uses same API as in standard library, so it has to be familiar to developers. Additionally it provides Semaphore, which is not implemented in std because of its rare use cases. Next code sample provides solution to classic assignment about philosophers and forks with Tokio Semaphore API.

C. Go

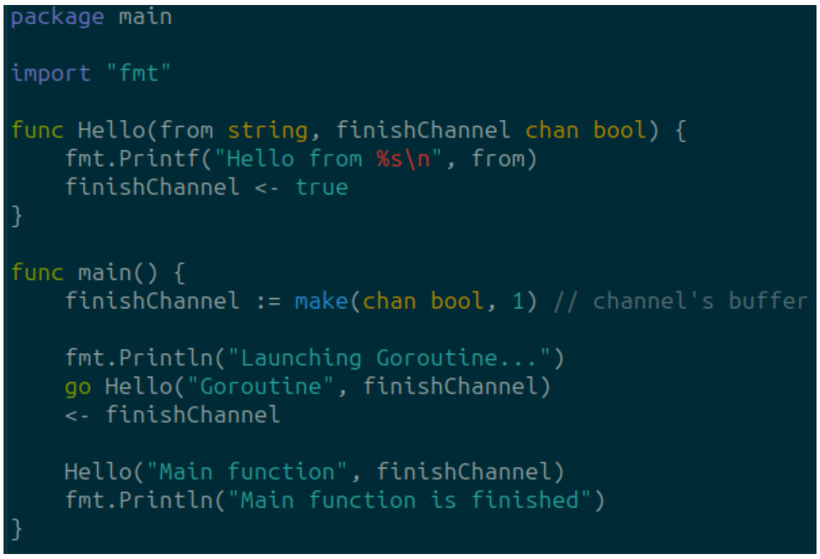
Go provides powerful API for both multithreaded and asynchronous execution - Goroutines. As coroutines in other languages, Goroutines have their own stack, but not their own control flow.

Go's runtime: Goroutines are managed by the implicit Go's runtime. As in other languages, this runtime uses scheduler to manage workload of goroutines. According to Kartik Khare's research [16], runtime depends on next three data structures:

- ***Goroutine's structure*** that holds its stack pointer, ID, cash and execution status
- ***OS Thread wrapper*** that stores pointer to all processing goroutines, running goroutine and reference to the scheduler
- ***Scheduler*** that internally is implemented as thread pool and goroutine pool - it manages both free and busy goroutines

Goroutines are very light with memory's resources. **Go's compiler knows how much memory does every function expose**, so, in order to run any function in Go inside a goroutine, **compiler can pre-allocate enough memory for the particular goroutine**.

Goroutine itself is not an explicit object, as Coroutines in Kotlin, they are bounded to the functions or methods that are called. According to Uday Hiwarale's article [17], Go's main function is not only start point of an app, but also the very first goroutine (or main goroutine), that is created by the runtime. Keyword *"go"* is used to execute function in another goroutine. Fig. 4.9 show an example of function executing in the separated goroutine.



```go
package main

import "fmt"

func Hello(from string, finishChannel chan bool) {
    fmt.Printf("Hello from %s\n", from)
    finishChannel <- true
}

func main() {
    finishChannel := make(chan bool, 1) // channel's buffer

    fmt.Println("Launching Goroutine...")
    go Hello("Goroutine", finishChannel)
    <- finishChannel

    Hello("Main function", finishChannel)
    fmt.Println("Main function is finished")
}
```

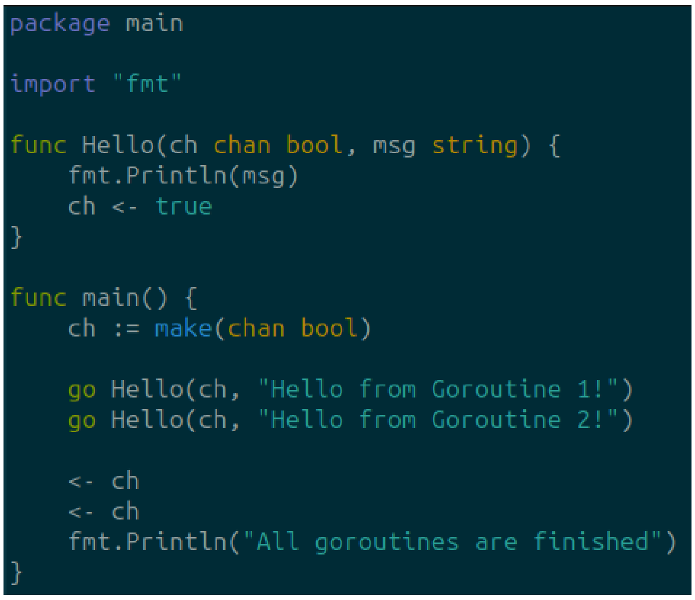
Fig. 4.9. Function Hello() launches in both main and new Goroutine

Channels: Goroutines are not used with async/await form, meaning that they cannot be explicitly awaited as coroutines. Instead, Go uses message-passing technique with ***channels*** - streams of data that can transfer it between goroutines.

Channel is a fundamental asynchronous unit in Go that allows to receive and send data between goroutines. Operations on channels are ”awaited”, meaning that goroutine will acquire ”sleeping” status until data is received. Sending goroutine also transferred to ”sleeping” state, until value is received in another goroutine. Fig. 4.9 demonstrates channels in action.

Channel is implemented as the queue of data. When value is added to the channel, it is pushed at the end of the queue, when it is removed, first value in queue (or top value) is taken. This allows to synchronize goroutines and main goroutine, meaning that principle of ”Continuation-Passing style” is also used for Go and Goroutines’ implementation. Fig. 4.10 illustrates synchronization of goroutines with channels.

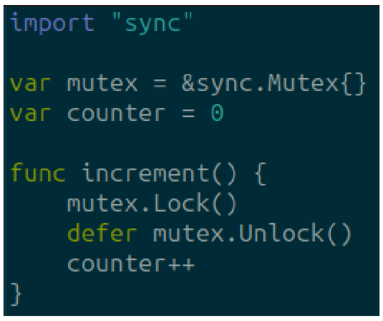
Synchronization: Channels allow to avoid ”shared mutability” and primitive synchronizations, making concurrency safe and efficient. However, message-passing interface is not always suitable for all tasks. Go provides several synchronization primitives to handle race conditions.



```go
package main

import "fmt"

func Hello(ch chan bool, msg string) {
    fmt.Println(msg)
    ch <- true
}

func main() {
    ch := make(chan bool)

    go Hello(ch, "Hello from Goroutine 1!")
    go Hello(ch, "Hello from Goroutine 2!")

    <- ch
    <- ch
    fmt.Println("All goroutines are finished")
}
```

Fig. 4.10. Channel as a synchronizer of goroutines

Mutex allows to lock and unlock goroutine. According to Lane Wagner's study [18], when mutex is locked, other goroutines that will try to lock will be blocked until unlock() is called. In other words, it is a classical mutex, but for goroutines' synchronization. Fig. 4.11 illustrates synchronization of goroutines with mutexes. Keyword 'defer' launches statement after function is finished.

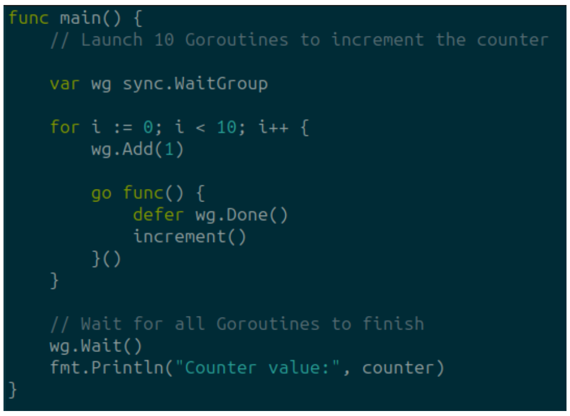


```go
import "sync"

var mutex = &sync.Mutex{}
var counter = 0

func increment() {
    mutex.Lock()
    defer mutex.Unlock()
    counter++
}
```

Fig. 4.11. Mutex in Go

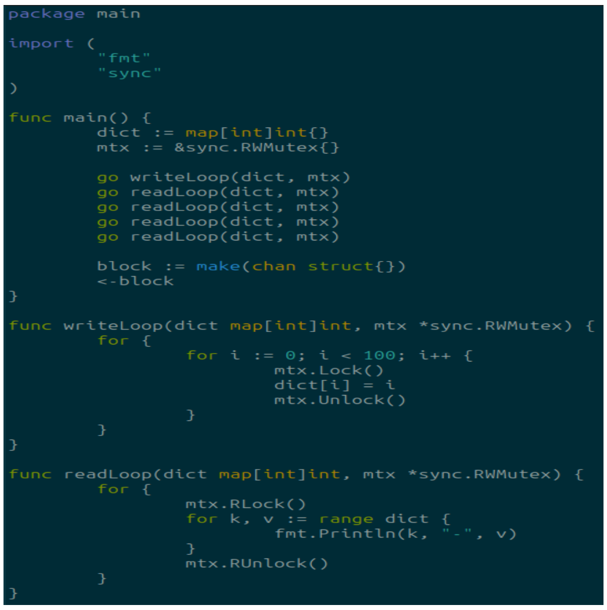
RWMutex is a classic *Read-Write lock*, that allows multiple readers or single writer. When writer appears, goroutines will be locked as in mutex, however, it allows to read data from multiple goroutines, if no active writers are present. Next code sample and Fig. 4.13 illustrates RWMutex in action.

WaitGroup is a sync primitive that allows to wait until all goroutines are finished. In some sense it has similar purpose with *Semaphore*: as semaphore, it counts the number of actions, and, as semaphore, it blocks until number of available slots is not decreased. However, WaitGroup has dynamic counting and unlocks after n goroutines are finished and called Done() method of WaitGroup, where n is the number of goroutines to await. Fig. 4.12 demonstrates how to wait for the completion of goroutines with WaitGroup.



```go
func main() {
    // Launch 10 Goroutines to increment the counter

    var wg sync.WaitGroup

    for i := 0; i < 10; i++ {
        wg.Add(1)

        go func() {
            defer wg.Done()
            increment()
        }()
    }

    // Wait for all Goroutines to finish
    wg.Wait()
    fmt.Println("Counter value:", counter)
}
```

Fig. 4.12. Await for all goroutines with WaitGroup



```go
package main

import (
        "fmt"
        "sync"
)

func main() {
        dict := map[int]int{}
        mtx := &sync.RWMutex{}

        go writeLoop(dict, mtx)
        go readLoop(dict, mtx)
        go readLoop(dict, mtx)
        go readLoop(dict, mtx)
        go readLoop(dict, mtx)

        block := make(chan struct{})
        <-block
}

func writeLoop(dict map[int]int, mtx *sync.RWMutex) {
        for {
                for i := 0; i < 100; i++ {
                        mtx.Lock()
                        dict[i] = i
                        mtx.Unlock()
                }
        }
}

func readLoop(dict map[int]int, mtx *sync.RWMutex) {
        for {
                mtx.RLock()
                for k, v := range dict {
                        fmt.Println(k, "-", v)
                }
                mtx.RUnlock()
        }
}
```

Fig. 4.13. RWMutex in Go

V Final words about Coroutines

Asynchronous execution with coroutines implemented within ***suspending execution*** and ***Continuation-Passing Style*** principles. Suspending execution allows to **suspend current function by switching control flow to the next function**. Suspending functions can be accessed again after suspense with ***Continuations***. When asynchronous step is finished, control flow switches back to the function with Continuation's call.

Coroutines are lightweight threads which may have their own stack memory, but **not their own control flow**. Coroutines are assigned to OS threads by the **coroutine scheduler (dispatcher)**, which usually manages both threads and coroutines together.

Chapter 5

Promises

JavaScript introduce its own approach for asynchronous API - Promises. ***Promise*** is an object that serves as an explicit manager for asynchronous tasks. Combining approaches of both coroutines and reactive observers, it provides powerful interface to handle parallel tasks.

I Promises runtime

As coroutines, promises' runtime is an event queue that stores functions that have to be executed. Since JavaScript was developed as Frontend language, no interface for an explicit multithreading is provided. It means that no synchronization mechanism is required, and event queue works as a typical queue of callbacks - it removes top callback and executes it.

II Promise as an object

Promise is a state holder of an asynchronous task. Internally, Promise is implemented as a wrapper over function which controls its state. According to Shailesh Shekhawat's study [19], Promises in JS/TS can have next three states:

- ***Fulfilled*** which indicates that promise is done and assigned function is completed with success

- ***Rejected*** which indicates that promise was canceled by an error or by user explicitly
- ***Pending*** which indicates that promise is still executing assigned function

First two states require additional optional callbacks to handle states.

Fulfilled callbacks are executed after a successful execution of assigned function. Promise may have multiple fulfilled callbacks that will be executed one after another. Each new callback constructs new Promise object that uses an argument taken as the result of a previous promise. This approach allows to manage declarative API similar to reactive observers. With method *then()* additional fulfilled callback will be assigned.

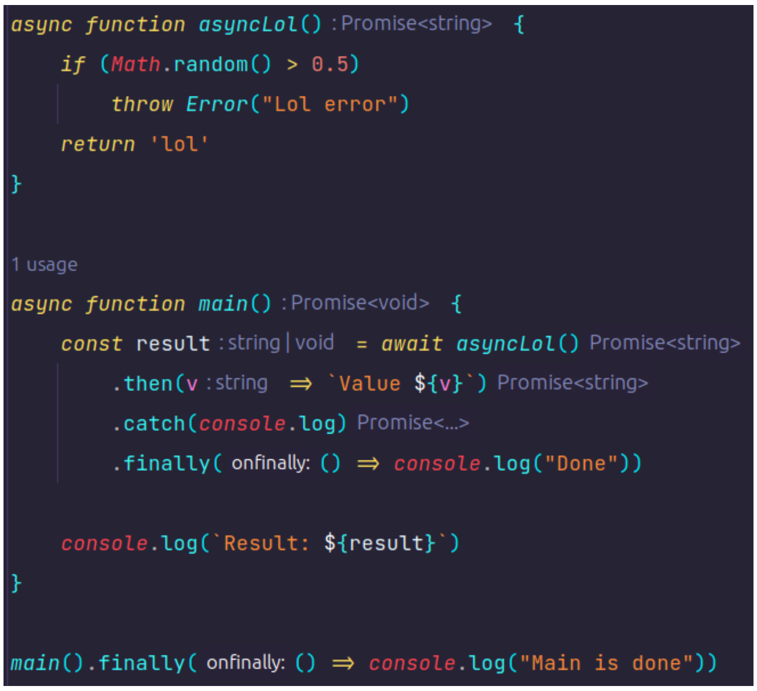
Reject callback is executed when error is occurred in one of the promises above. After callback is executed, **undefined is returned**. With TypeScript's type safety further manipulations with data can be applied: user can introduce new fulfiled callbacks and reject callbacks, but the resulting value will be *T—undefined*, which means that developer has to check for an error in case if he wishes to manage promises' chain further.

All in all, promises provides declarative API which is similar to try/catch/finally block, but allows asynchronous execution. Fig. 5.1 provides an example of promise that can either finish with an error or with success. Fig. 5.2-5.3 illustrates possible outputs after both scenarios.

III Await to complete

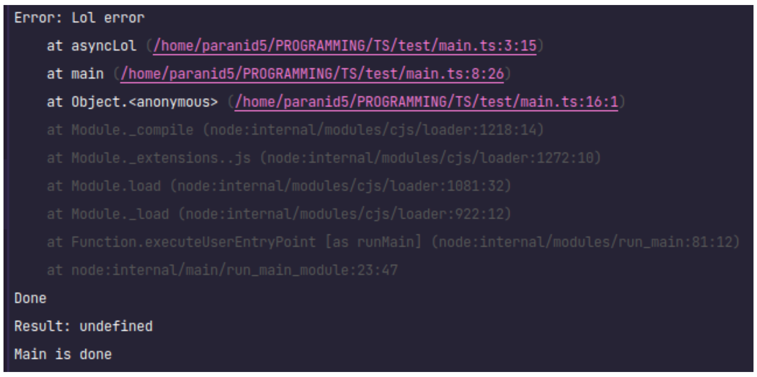
Similar to coroutines, Promises provides async/await API to manage the result in a synchronous way. According to the book by Dmitry Sheiko et al. [20], Promises use CPS method explicitly to manage next continuations (wrapped as Promises) and handle errors (which are also continuations wrapped in Promises).

When control flow reaches *"await"* keyword call, it suspends until result is received, which allows to update DOM without any blocks. "Await" keyword can be used only inside *"async"* functions. Since JavaScript disallows any blocking technique, async functions do not have to be always awaited and can finished whenever they are completed or with Promises' callbacks discussed above.



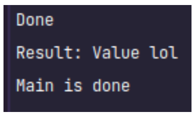
```
async function asyncLol() :Promise<string>  {
    if (Math.random() > 0.5)
        throw Error("Lol error")
    return 'lol'
}

1 usage
async function main() :Promise<void>  {
    const result :string | void  = await asyncLol() Promise<string>
        .then(v :string  ⇒ `Value ${v}`) Promise<string>
        .catch(console.log) Promise<...>
        .finally( onfinally: () ⇒ console.log("Done"))

    console.log(`Result: ${result}`)
}

main().finally( onfinally: () ⇒ console.log("Main is done"))
```

Fig. 5.1. Promise that can finish with an error or success



```
Error: Lol error
    at asyncLol (/home/paranid5/PROGRAMMING/TS/test/main.ts:3:15)
    at main (/home/paranid5/PROGRAMMING/TS/test/main.ts:8:26)
    at Object.<anonymous> (/home/paranid5/PROGRAMMING/TS/test/main.ts:16:1)
    at Module._compile (node:internal/modules/cjs/loader:1218:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1272:10)
    at Module.load (node:internal/modules/cjs/loader:1081:32)
    at Module._load (node:internal/modules/cjs/loader:922:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:81:12)
    at node:internal/main/run_main_module:23:47
Done
Result: undefined
Main is done
```

Fig. 5.2. On error thrown and catch is called



```
Done
Result: Value lol
Main is done
```

Fig. 5.3. Success

Result of "await" operation is always the result of the Promise. In other words, if no catch() callback was applied, TypeScript's type system will consider the result as T, where T is the return type. In case of error, exception will be thrown. If catch() callback was provided, TypeScript's type system will consider result as T—*undefined*, which is either success or an error. It is up to developer to check the return value. Fig. 5.1 illustrates usage of "async/await" mechanism.

Chapter 6

Final words before you go

Together we have studied numerous methods to implement asynchronous workflow in the application. Reactive observers can significantly improve the performance of IO operations or can be used as an architectural pattern for an application that depends on the reactive execution (such as Web or UI applications). Suspending execution provides asynchronous API to use the result in the same control flow as in the sequential execution. Coroutines proved their worth in many scenarios and suspending execution itself together with "continuation-passing style" introduce numerous opportunities to implement many ideas, including coroutines, reactive paradigm and promises. At the same time, idea of Promises allows to combine advantages of both reactive and suspending approaches, giving developer freedom to choose.

Asynchronous technique is a powerful tool that allows your application to be as smart as possible. I hope that now you will not be afraid of complex structure with reactive observers and your brain won't explode from multiple coroutines that launches concurrently. Now you know these "asynchronous beasts" in person and ready to use them with all their power!

References

[1] R. Elizarov, “Blocking threads, suspending coroutines,” Nov. 2018. [Online]. Available: https://medium.com/@elizarov/blocking-threads-suspending-coroutines-d33e11bf4761.

[2] N. Skelton, “Unit testing kotlin flow,” *Google Developer Experts*, Feb. 2021. [Online]. Available: https://medium.com/google-developer-experts/unit-testing-kotlin-flow-76ea5f4282c5.

[3] D. Santiago, “Threading models in coroutines and android sqlite api,” Jul. 2019. [Online]. Available: https://medium.com/androiddevelopers/threading-models-in-coroutines-and-android-sqlite-api-6cab11f7eb90.

[4] K. Patel, “What is reactive programming?,” Dec. 2016. [Online]. Available: https://medium.com/@kevalpatel2106/what-is-reactive-programming-da37c1611382.

[5] K. Verma, “Asynchronous programming with thread pools,” *Level Up Coding*, Nov. 2019. [Online]. Available: https://medium.com/gitconnected/asynchronous-programming-with-thread-pools-e42d6bacd171.

[6] E. G. et al., “Design patterns: Elements of reusable object-oriented software,” Oct. 1994.

[7] avocadi, “What is epoll?,” May 2022. [Online]. Available: https://medium.com/@avocadi/what-is-epoll-9bbc74272f7c.

[8] V. Drobushkov, “Decoding handler and looper in android,” *ProAndroidDev*, Dec. 2019. [Online]. Available: https://proandroiddev.com/decoding-handler-and-looper-in-android-d4f3f2449513.

[9] C. Escoffier, "5 things to know about reactive programming," 2017. [Online]. Available: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming.

[10] J. E. Picó, "Going deep on flows channels — part 1: Streams," Jun. 2022. [Online]. Available: https://proandroiddev.com/going-deep-on-flows-channels-part-1-streams-5ae8b8491ac4.

[11] R. Elizarov, "Cold flows, hot channels," Apr. 2019. [Online]. Available: https://elizarov.medium.com/cold-flows-hot-channels-d74769805f9.

[12] digitake, "On the continuation-passing style and its role in fp," *LambdaSide*, Jun. 2018. [Online]. Available: https://medium.com/lambdaside/on-the-continuation-passing-style-and-its-role-in-fp-d07f151b6c5c.

[13] S. Vergauwen, "Continuation in kotlin," Nov. 2021. [Online]. Available: https://nomisrev.github.io/continuation-monad-in-kotlin/.

[14] Karthik, "Swift data race vs. race condition," Oct. 2021. [Online]. Available: https://medium.com/p/5d92c389a3ab.

[15] A. Shekhar, "Dispatchers in kotlin coroutines," Oct. 2022. [Online]. Available: https://medium.com/@amitshekhar/dispatchers-in-kotlin-coroutines-7f1626b23d9a.

[16] K. Khare, "Why goroutines are not lightweight threads?," Mar. 2018. [Online]. Available: https://medium.com/codeburst/why-goroutines-are-not-lightweight-threads-7c460c1f155f.

[17] U. Hiwarale, "Anatomy of goroutines in go -concurrency in go," Nov. 2018. [Online]. Available: https://medium.com/rungo/anatomy-of-goroutines-in-go-concurrency-in-go-a4cb9272ff88.

[18] L. Wagner, "Golang mutexes — what is rwmutex for?," Mar. 2020. [Online]. Available: https://medium.com/bootdotdev/golang-mutexes-what-is-rwmutex-for-5360ab082626.

[19] S. Shekhawat, "How javascript promises actually work from the inside out," Feb. 2019. [Online]. Available: https://www.freecodecamp.org/news/how-javascript-promises-actually-work-from-the-inside-out-76698bb7210b/.

[20] D. S. et al., "Javascript unlocked," Dec. 2015. [Online]. Available: https://subscription.packtpub.com/book/programming/9781785881572/5/ch05lvl1sec31/continuation-passing-style.