Asynchronous programming

Hadi Hariri
Internet Explorer has encountered a problem and needs to close. We are sorry for the inconvenience.

If you were in the middle of something, the information you were working on might be lost.

Please tell Microsoft about this problem.
We have created an error report that you can send to help us improve Internet Explorer. We will treat this report as confidential and anonymous.

To see what data this error report contains, click here.

Send Error Report  Don't Send
Approaches
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}

fun preparePost(): Token {
    // makes request & blocks thread
    return token
}
Threading

- **Complex** - state mutations, UI threads, debugging
- **Unviable** - single threaded environments
- **Limited** - underlying operating system
- **Expensive** - context switches
callbacks

```kotlin
fun postItem(item: Item) {
    preparePostAsync { token ->
        submitPostAsync(token, item) { post ->
            processPost(post)
        }
    }
}

fun preparePostAsync(cb: (Token) -> Unit) {
    // makes request & return immediately
    // arranges callback to be called later
}
```
Callbacks

• **Complex** - Error handling

• **Unreadable** - Callback Hell
Promises and Futures

```kotlin
fun postItem(item: Item) {
    preparePostAsync()
    .thenCompose(token ->
        submitPostAsync(token, item)
    )
    .thenAccept(post ->
        processPost(post)
    )
}

fun preparePostAsync(): Promise<Token> {
    // makes result & return a promise that
    return promise // .... is completed later
}
```
Promises and Futures

- **Complex** - error handling can be complicated
- **Paradigm** - new ways of combining, learning new combinators, etc.
interface Observable<out T> {
    fun Subscribe(observer: Observer<T>)
}

interface Observer<in T> {
    fun onNext(t: T)
    fun onError(e: Exception)
    fun onCompleted()
}
Reactive Extensions

- **Paradigm** - new ways of combining, learning new combinators, etc.

Everything is a stream. And it’s observable.
Kotlin Co-routines
Do async, the same way you’d do sync
In essence...

- Same way of thinking for sync and async
- Same way of doing things (exception handling, loops, etc.)
- Same constructs
Coroutines

- Suspensible Computation
  - A function can be suspended, and later resumed
- Co-operative multitasking a.k.a. non-preemptive
- State machine with CPS
- Lightweight threads
How they look

```kotlin
fun postItem(item: Item) {
    launch(CommonPool) {
        1. val token = preparePost()
        2. val post = submitPost(token, item)
        3. processPost(post)
    }
}
```

**suspension function**

```kotlin
suspend fun preparePost(): Token {
    // makes request & suspends coroutine
    return suspendCoroutine { /* ... */ }
}
```
How they work

Kotlin

```kotlin
suspend fun submitPost(token: Token, item: Item): Post {...}
```

Java/JVM

```java
Object submitPost(Token token, Item item, Continuation<Post> cont) {...}
```

```java
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```
How they’re implemented

Kotlin

```kotlin
... val token = preparePost()
... val post = submitPost(token, item)
... processPost(post)
```

Java/JVM

```java
switch (cont.label){
    case 0:
        cont.label = 1;
        preparePost(cont);
        break;
    case 1:
        Token token = (Token) prevResult;
        cont.label = 2;
        submitPost(token, item, cont);
        break;
    case 2:
        Post post = (Post) prevResult;
        processPost(post);
}
```
Let’s see some code!
Register early and get the best discounts!