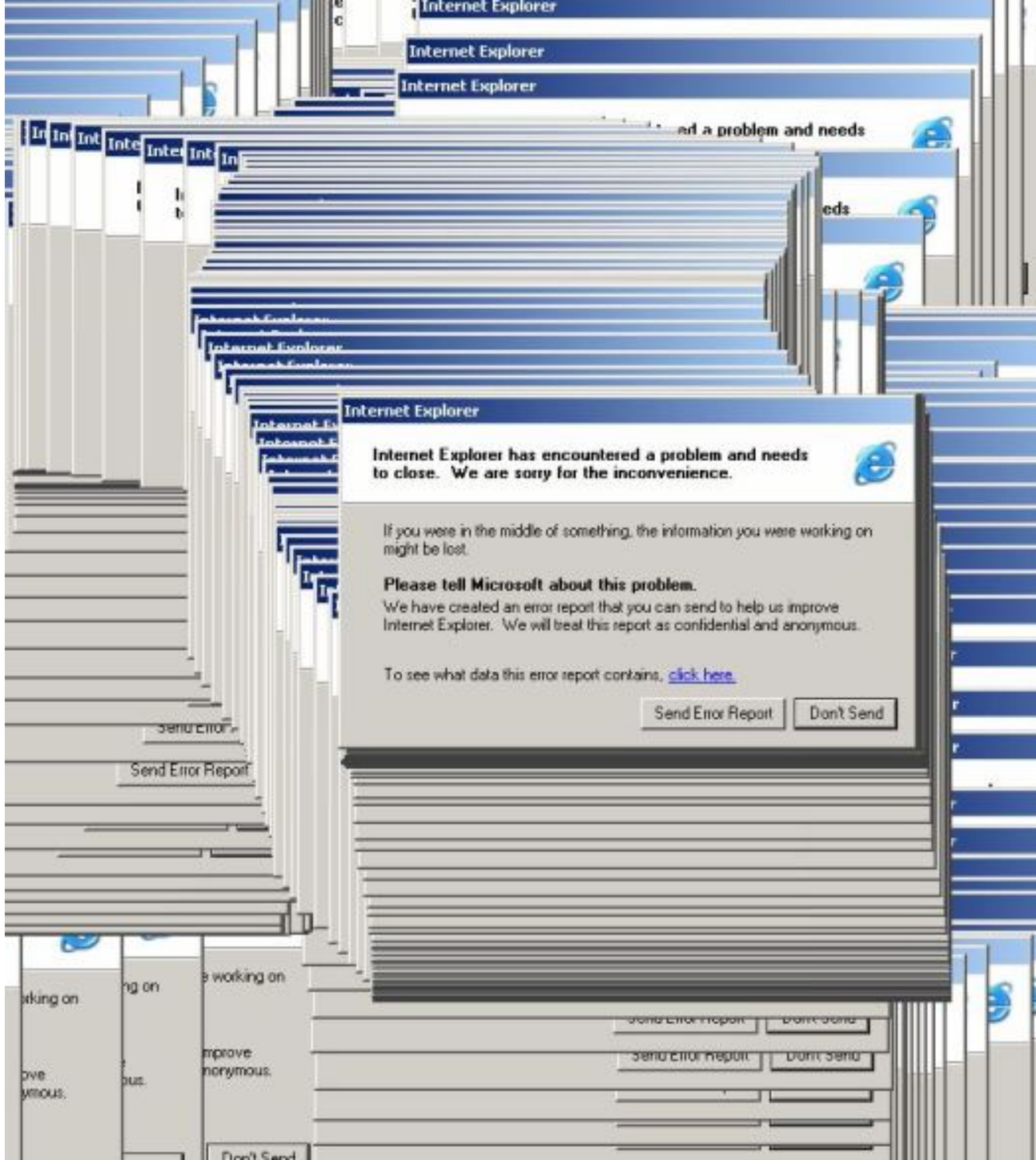
Asynchronous Programming

It’s hard...



Possible solutions

- Threading
- Callbacks
- Promises
- Futures
- Reactive Extensions

Threading

```
fun postItem(item: Item) {
    val token = preparePost()
    val post = submitPost(token, item)
    processPost(post)
}
```

Threaded

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

Threading

- Threads are expensive - context switches
- Threads are not possible - single threaded targets
- Threads are not desirable - state mutations, UI threads

Callbacks

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

async

callback

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

Callbacks

- Error handling can be complicated

- Callback hell a.k.a. tilted Christmas Tree

Promises, Futures and Rx

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

composable

promise

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

Futures, Promises and Rx

- Error handling can be complicated
- Thinking of different constructs on different platform
- Changing the way of thinking for sync vs async (move to streams)

Kotlin Coroutines

Coroutines

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

explicit coroutine context

natural signature

suspending function

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

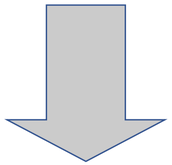
Coroutines

- Same way of thinking for sync and async
- Same way of doing things (exception handling, loops, etc.)
- Same constructs
- Lightweight threads
- Long time existing concepts

How they work

Kotlin

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



Java/JVM

callback

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

```
public interface Continuation<in T> {
    public val context: CoroutineContext
    public fun resume(value: T)
    public fun resumeWithException(exception: Throwable)
}
```

How they work

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);
    }
}
```

Constructs

- **Builders (launch, runBlocking, async)**: regular world to coroutine world
- **Suspending functions (suspend):** from coroutine to coroutine
- **suspendCoroutine:** coroutine to callback