

# Lecture 10

COMP 3717- Mobile Dev with Android Tech

# Asynchronous programming

- The execution of one task isn't dependent on another (aka. non blocking)
- All our code in this course so far has run *synchronously*
  - The execution of each operation depends on completing the one before it
- Asynchronous operations allow a program to be more efficient
  - E.g., Making a request to a server without freezing the screen

# Asynchronous tools

- *AsyncTask*
  - Deprecated, too unstable
- *Executors* and *Futures*
  - Recommended for java
- RxJava & RxKotlin
  - Popular library
- *Coroutines*
  - Recommended for Kotlin

# Coroutines

- A concurrency design pattern
- A coroutine can be compared to a thread but are different at the lower level
- At the lower level, a coroutine saves state and runs it at later time
  - Uses *continuations* under the hood; a special type of callback
  - This can be done on a single thread

# Coroutines (cont.)

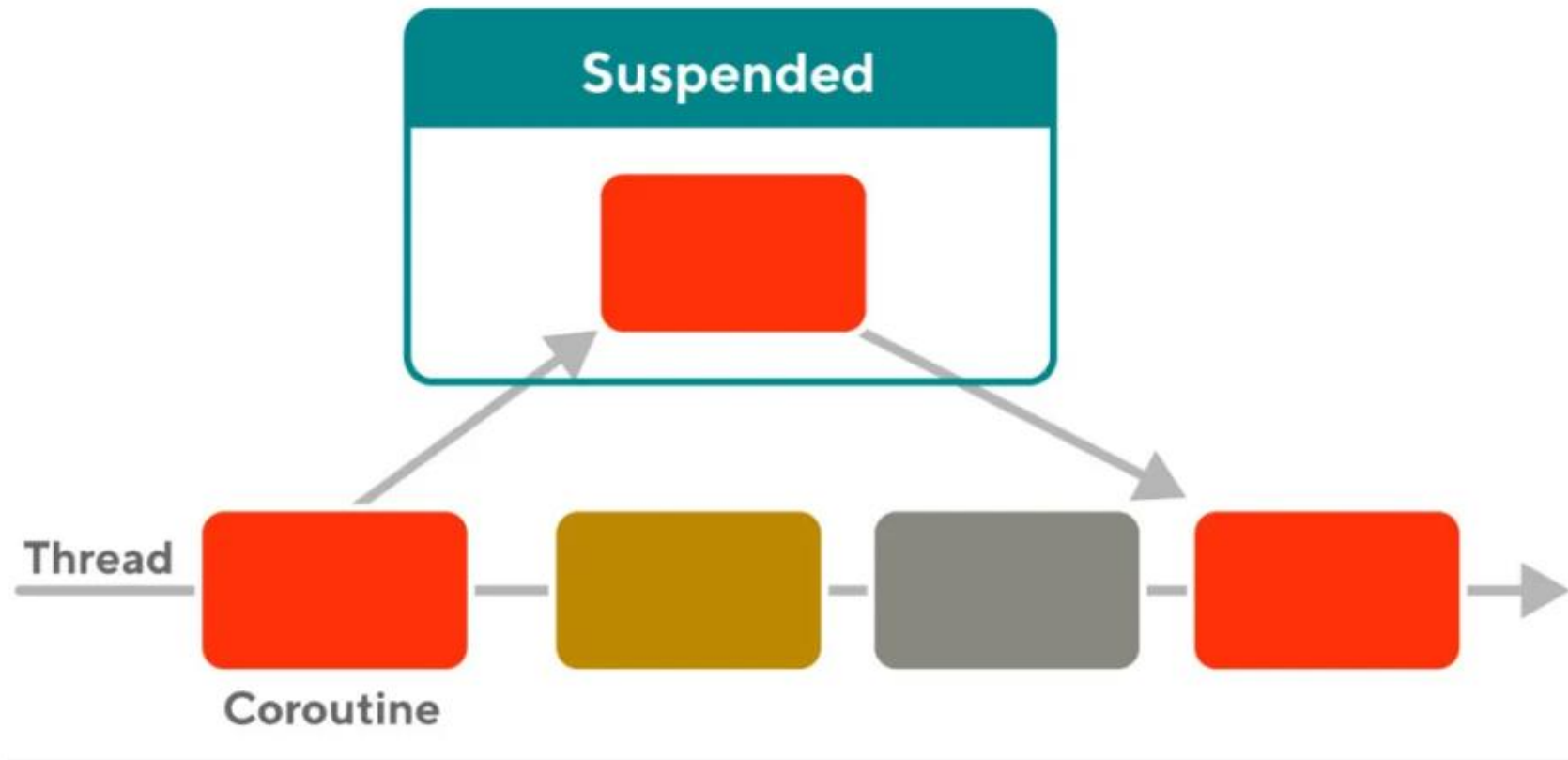
- The way a coroutine saves state it through *suspending* functions
- To make a function suspending we use the *suspend* modifier

```
suspend fun mySuspendingFunction(){
```

# Coroutines

- When a coroutine calls a suspending function, the coroutine is *suspended*
- Once suspended its state gets saved and the regular flow of operations continue (aka. non-blocking)
- When the suspended operation completes (e.g. an http request), its state is restored back with the regular flow of operations

# Coroutines




# Coroutines

- You call suspend functions only from other suspend functions or directly within a coroutine
- To create a coroutine we use a coroutine builder
  - runBlocking
    - Blocks current thread, usually used at top level of application
  - launch
    - Non-blocking; returns a *Job* object and does not provide a result
  - async
    - Non-blocking; returns a *Deferred Job* which provides a result

# Kotlin Coroutines (cont.)

- A *CoroutineScope* defines the lifetime of a coroutine and its context
  - A *CoroutineContext* defines how a coroutine is executed
  - Every coroutine will need a *CoroutineScope*

```
runBlocking{ this: CoroutineScope  
  
}
```



- Usually, we must provide our own *CoroutineScope*, but it is created internally using *runBlocking*

# Kotlin Coroutines (cont.)

- **delay**
  - A suspending function that delays the coroutine for a given duration

```
fun main() {  
    runBlocking {  
        print("The sponge...")  
        delay( timeMillis: 1000L)  
        println("is back!")  
    }  
}
```



- Since *runBlocking* blocks the current thread, the above is like using *Thread.sleep*

# Kotlin Coroutines (cont.)

- Here I am using the same logic but with my own suspend function

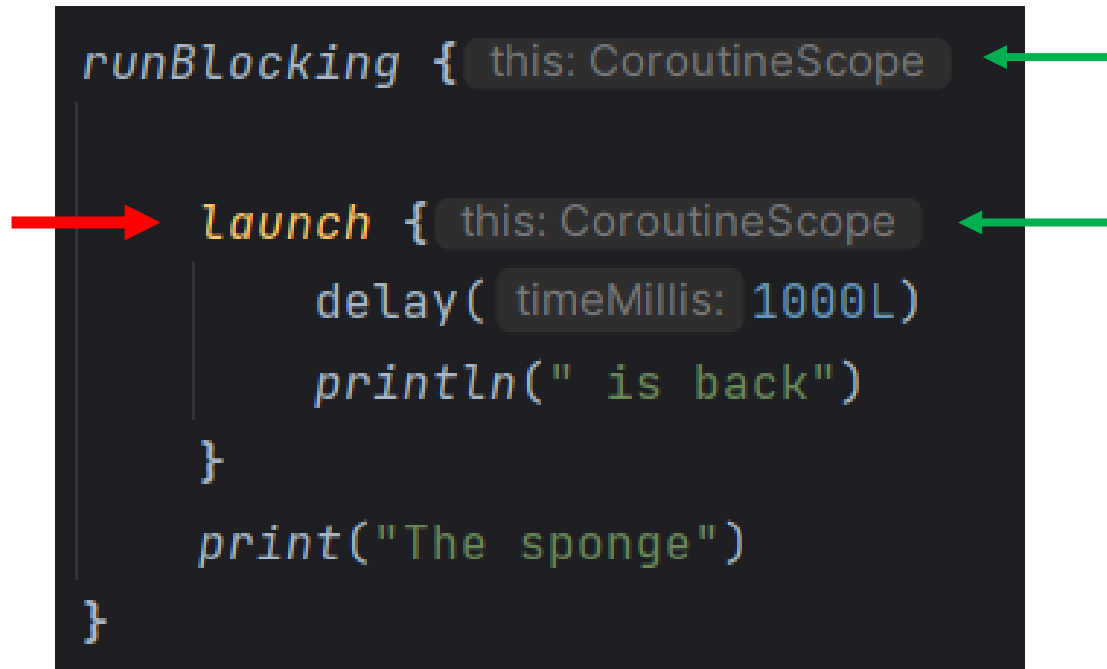
```
fun main() {  
    runBlocking {  
        someFun()  
    }  
}
```

```
suspend fun someFun(){  
    print("The sponge...")  
    delay(timeMillis: 1000L)  
    println("is back!")  
}
```

# Kotlin Coroutines (cont.)

- The **launch** coroutine builder creates a non-blocking coroutine
  - It can only be called with a *CoroutineScope*
  - In this case, it **inherits runBlocking's *CoroutineScope***

```
runBlocking { this: CoroutineScope  
    launch { this: CoroutineScope  
        delay( timeMillis: 1000L)  
        println(" is back")  
    }  
    print("The sponge")  
}
```

A diagram illustrating the CoroutineScope inheritance. A red arrow points from the `launch` builder to the `this: CoroutineScope` parameter in its lambda block. A green arrow originates from the `this: CoroutineScope` parameter in the `runBlocking` block and points to the `this: CoroutineScope` parameter in the `launch` block, indicating that the `launch` coroutine inherits the `CoroutineScope` from the `runBlocking` block.

# Kotlin Coroutines (cont.)

- We could also move all of this in to a suspend function and provide our own *CoroutineScope* using the *coroutineScope*

```
fun main() {  
    runBlocking {  
        someFun()  
    }  
}
```



```
suspend fun someFun(){  
    coroutineScope {  
        launch {  
            delay( timeMillis: 1000L)  
            println("is back!")  
        }  
        print("The sponge...")  
    }  
}
```

# Kotlin Coroutines (cont.)

- The returned *Job* object can be used to manage our coroutine
  - *join*: Waits until coroutine is finished

```
runBlocking { this: CoroutineScope

    val job = launch { this: CoroutineScope
        delay( timeMillis: 1000L)
        print(" is")
    }

    print("The sponge")
    job.join() ←
    println(" back")
}
```

# Kotlin Coroutines (cont.)

- We use *async* over *launch* when we want a returned result
  - **await**: Waits until coroutine is finished and provides a result

```
runBlocking { this: CoroutineScope

    val deferredJob = async { this: CoroutineScope
        delay( timeMillis: 1000L)
        " is" ^async
    }

    print("The sponge")
    print(deferredJob.await())
    println(" back")
}
```



- In this example, *deferredJob* returns a *Deferred* of type **String**

# Kotlin Coroutines (cont.)

- launch and async can also be **cancelled**

```
val job = launch {  
    println("The sponge is on his way...")  
    repeat(times: 1000){  
        println("waiting...")  
        delay(timeMillis: 1000L)  
    }  
}  
delay(timeMillis: 5000L)  
→ job.cancel()  
print("The sponge has arrived!")
```

# HTTP Requests

- A common asynchronous task is to make an HTTP request
  - E.g., CONNECT, GET, POST, etc. to an API
- An HTTP request can take time, especially if the network is poor
  - We don't want to freeze our app (aka. block the main thread)
- There are many ways to make HTTP requests
  - Use the java standard library
    - *HttpURLConnection*
  - Use a third-party library such as Ktor

# Serialization and deserialization

- Serialization
  - Converting your data into a format that can be stored or transmitted
    - E.g., Across a network
- Deserialization
  - Converting that format back into a data structure
- Serializing data into JSON format is popular and simple
  - JSON: Text-based format that uses key-value pairs and arrays

# Allow internet access for your app

- To give your app internet access you need to update your system permissions
- You do this in *AndroidManifest.xml* using the `<uses-permission>` element

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    — <uses-permission android:name="android.permission.INTERNET" /> —

    <application
        android:allowBackup="true"
```

# Ktor

- Ktor is built using Kotlin Coroutines and provides us with
  - A client to make HTTP requests
  - JSON Serialization and deserialization
- To use Ktor we need to add a few dependencies

```
dependencies {  
  
    implementation("io.ktor:ktor-client-android:3.0.1")  
    implementation("io.ktor:ktor-client-content-negotiation:3.0.1")  
    implementation("io.ktor:ktor-serialization-gson:3.0.1")  
}
```

## Ktor (cont.)

- Ktor's *HttpClient* is built using Coroutines

```
private val client = HttpClient{  
  
}
```

- *HttpClient* is the entry point for creating HTTP requests

# Ktor (cont.)

- Ktor has many JSON serializers but the one we will use is *GSON*
  - Here we use the *install* function to use the specific plugin

```
val client = HttpClient{ this: HttpClientConfig<*>  
    install(ContentNegotiation){ this: ContentNegotiation.Config  
        → gson()  
    }  
}
```

# Ktor (cont.)

- We can add specific headers for our request here too
  - Ex. Adding an *Authorization* header and an API key in our request

```
val client = HttpClient{  
  
    install(ContentNegotiation){...}  
  
    defaultRequest {  
        header(HttpHeaders.Authorization, "Bearer $API_KEY")  
    }  
}
```

# Consuming an API

- Application programming interface (aka. API)
  - Allows a way for two or more computers to communicate
- The API we are consuming is from the Art institute of Chicago
  - <http://api.artic.edu/docs/>
- Most modern APIs provide their data in JSON format
  - [https://api.artic.edu/api/v1/artworks?fields=id,title,image\\_id](https://api.artic.edu/api/v1/artworks?fields=id,title,image_id)

# Consuming an API (cont.)

- When consuming an API, it's first important to determine what data you want in your app
- Many APIs provide far more data than is needed for the program requesting the information
- Once you know the specific data, you should create the data model

# Consuming an API (cont.)

- There are automated tools for creating a data models such as
  - Online JSON formatters
  - JetBrains plugin
    - JSONToKotlinClass
- You can also just create the model yourself if the API is simple enough and you know how to read JSON

# Understanding JSON

- A JSON is made up of keys and values
  - A key is always a string
  - A value can be
    - An object which is wrapped in { }
    - An array which is wrapped in [ ]
    - A string, number or boolean

```
{
  "results": [
    {
      "bill_id": "s1917-113",
      "chamber": "senate",
      "congress": 113,
      "number": 62,
      "question": "On Passage of the Bill S. 1917",
      "required": "1/2",
      "result": "Bill Passed",
      "roll_id": "s62-2014",
      "roll_type": "On Passage of the Bill",
    }
  ]
}
```

**GUESS WHAT ?**



**BREAK TIME !**

memegenerator.net

# Consuming an API (cont.)

- Here is a simple data model for the JSON I want to use
  - [https://api.artic.edu/api/v1/artworks?fields=id,title,image\\_id](https://api.artic.edu/api/v1/artworks?fields=id,title,image_id)
  - Notice I don't need to model everything in the JSON

```
data class Art (  
    val pieces: List<ArtPiece>  
)  
  
data class ArtPiece(  
    val id: String,  
    val title: String,  
    val image: String?  
)
```

# Consuming an API (cont.)

- The **variable name** needs to match the name in the JSON

```
data class Art(  
    @SerializedName("data")  
    val pieces: List<ArtPiece>  
)  
  
data class ArtPiece(  
    val id: String,  
    val title: String,  
    @SerializedName("image_id")  
    val image: String?  
)
```

```
{  
  "data": [  
    {  
      "id": 270374,  
      "title": "C  
300-2e55bcad0d94",  
      "image": "f1" } ]
```

- If you want to have a different variable name, then use *@SerializedName* and provide the name that matches the JSON

# Consuming an API (cont.)

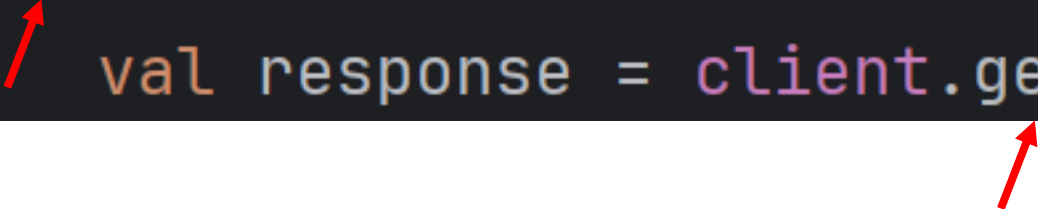
- Sometimes you have multiple endpoints
  - Endpoints are URLs that represent specific resources or actions in an API
- Its good practise to make your endpoints constants

```
const val BASE_URL = "https://api.artic.edu/api/v1"  
const val ARTWORKS = "${BASE_URL}/artworks"  
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id"
```

# Consuming an API (cont.)

- We can now call the HTTP request GET
  - Since this is data access logic, we should put it in a Repository

```
suspend fun getArtwork(): Art{  
    val response = client.get(FIELDS)
```



- Ktor's *client.get* is a suspend function so we need to put this in another suspend function

# Consuming an API (cont.)


- Next, we will make use of GSON
- The *HttpResponse.body* provides us with the **Json** in type *JsonObject*

```
suspend fun getArtwork(): Art{  
    val response = client.get(FIELDS)  
    val json = response.body<JsonObject>().toString()  
}
```

# Consuming an API (cont.)

- Here we are using GSON to deserialize the Json object into a new instance of our **data model**

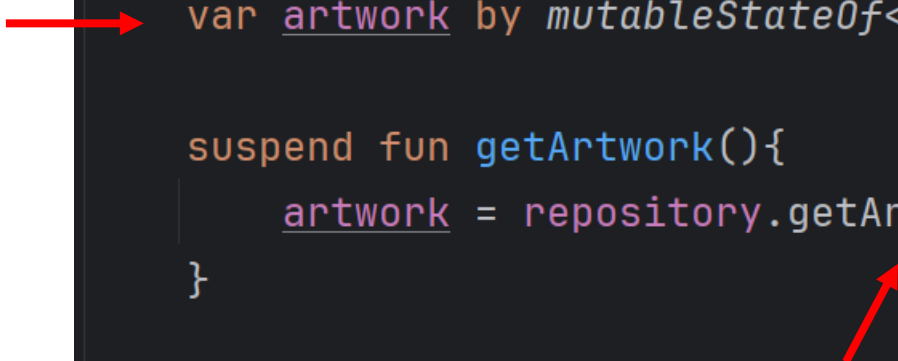
```
suspend fun getArtwork(): Art{  
    val response = client.get(FIELDS)  
    val json = response.body<JsonObject>().toString()  
    return Gson().fromJson(json, Art::class.java)  
}
```



# Consuming an API (cont.)

- Once our repository is set up, we can set up our state holder

```
class ArtState(private val repository: ArtRepository) {  
    → var artwork by mutableStateOf<Art?>(value: null)  
  
    suspend fun getArtwork(){  
        artwork = repository.getArtwork()  
    }  
}
```

Two red arrows are present: one points from the left to the `artwork` variable in the `mutableStateOf` declaration, and the other points from the bottom right to the `suspend fun` keyword.

- `getArtwork` is **suspending** so it needs to go in another suspend function

# Consuming an API (cont.)

- LaunchedEffect is a composable AND a coroutine
  - Useful for running suspend functions in the scope of a composable

```
class MainActivity : ComponentActivity() {  
  
    private val artRepository by lazy{  
        ArtRepository(client)  
    }  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        enableEdgeToEdge()  
        setContent {  
            val artState = ArtState(artRepository)  
            LaunchedEffect(artState) {  
                artState.getArtwork()  
            }  
        }  
    }  
}
```

# Consuming an API (cont.)

- *LaunchedEffect* will re-launch if its **key** changes
- *LaunchedEffect* will not re-launch if recomposed and the **key** doesn't change

```
setContent {  
  
    val artState = ArtState(artRepository)  
  
    LaunchedEffect(artState) {  
        artState.getArtwork()  
    }  
  
    MainContent(artState)  
  
}
```



# Consuming an API (cont.)

- If your API provides images, you first need to obtain the correct URL
- For the Chicago Art institute API, the format is below
  - Docs: <https://api.artic.edu/docs/#images>

```
const val FIELDS = "${ARTWORKS}?fields=id,title,artist_display,image_id  
const val IMAGE = "https://www.artic.edu/iiif/2/%s/full/843,/0/default.jpg"
```

# Coil

- Coil is an image loader library for jetpack compose

```
dependencies {  
  
    implementation("io.coil-kt.coil3:coil-compose:3.0.3")  
    implementation("io.coil-kt.coil3:coil-network-okhttp:3.0.3")  
}
```

- It makes our life easy

```
AsyncImage(  
    model = IMAGE.format(pieces?.get(it)?.image),  
    contentDescription = null  
)
```

# Consuming an API (cont.)

- Display your data however you want

Black-and-White Storage  
Jar with Abstract Geometric  
Motifs

Acoma  
Acoma Pueblo, New Mexico,  
United States

1890s



The Elements (Furnishing  
Fabric)

Designed by Bonaventure M.  
Lebert (French, 1759-1836)  
Manufactured by Hartmann et  
Fils (French, founded 1776)  
France, Nantes

1810/20



Bottle Rack (Porte-Bouteilles)

Marcel Duchamp  
American, born France,  
1887–1968

1914/1959



