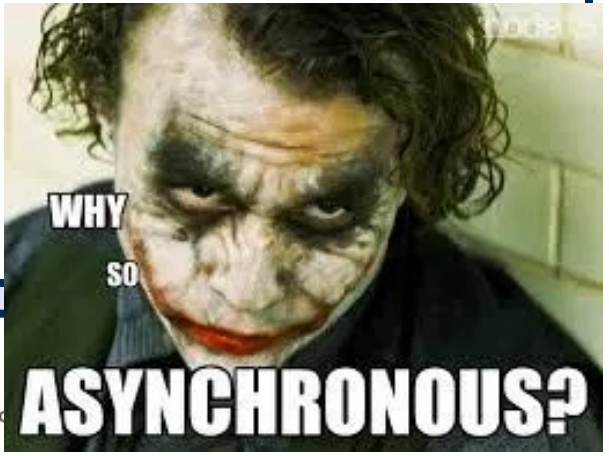
Mobile Applications CSCI 448 Lecture 20

Networking:

Asynchronous Tasks via WorkManager



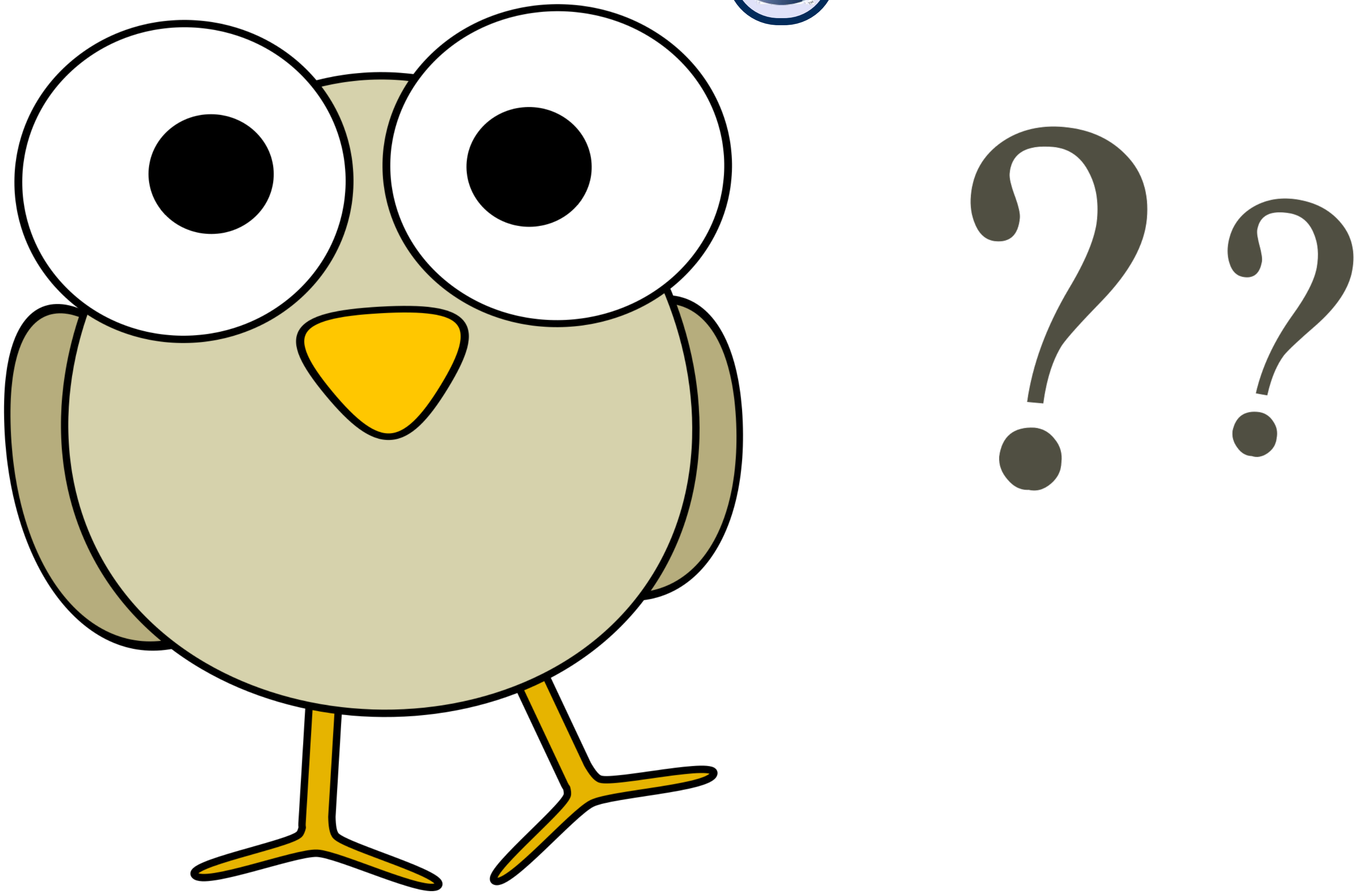
"The key to time management is efficiency. I do all of my pointing on Monday, Wednesday and Friday, then all of my clicking on Tuesday and Thursday."

Previously in CSCI 448

- Room Database Stack
  - View
  - View Model / ViewModel & ViewModelFactory
  - Repository
  - DAO
  - Room DB

- Singleton & Façade Design Patterns

Questions?



Learning Outcomes For Today

- Create an app that accesses the network via WorkManager
- Discuss how WorkManager works and concerns that arise

On Tap For Today

- A Background Thread
- WorkManager
- Practice

On Tap For Today

- A Background Thread
- WorkManager
- Practice

NETWORKING!

- Connect to the internet via
  - WiFi
  - 3G (4G) [5G] <6G> {7G}

- MUST be done on a background thread

Why a Background Thread?

- Networking sends request to web server
  - Who knows when, or if, a response will come
  - Do not want UI frozen waiting for web response

- If you try to network from main thread you get

  android.os.NetworkOnMainThreadException

Types of Background Work

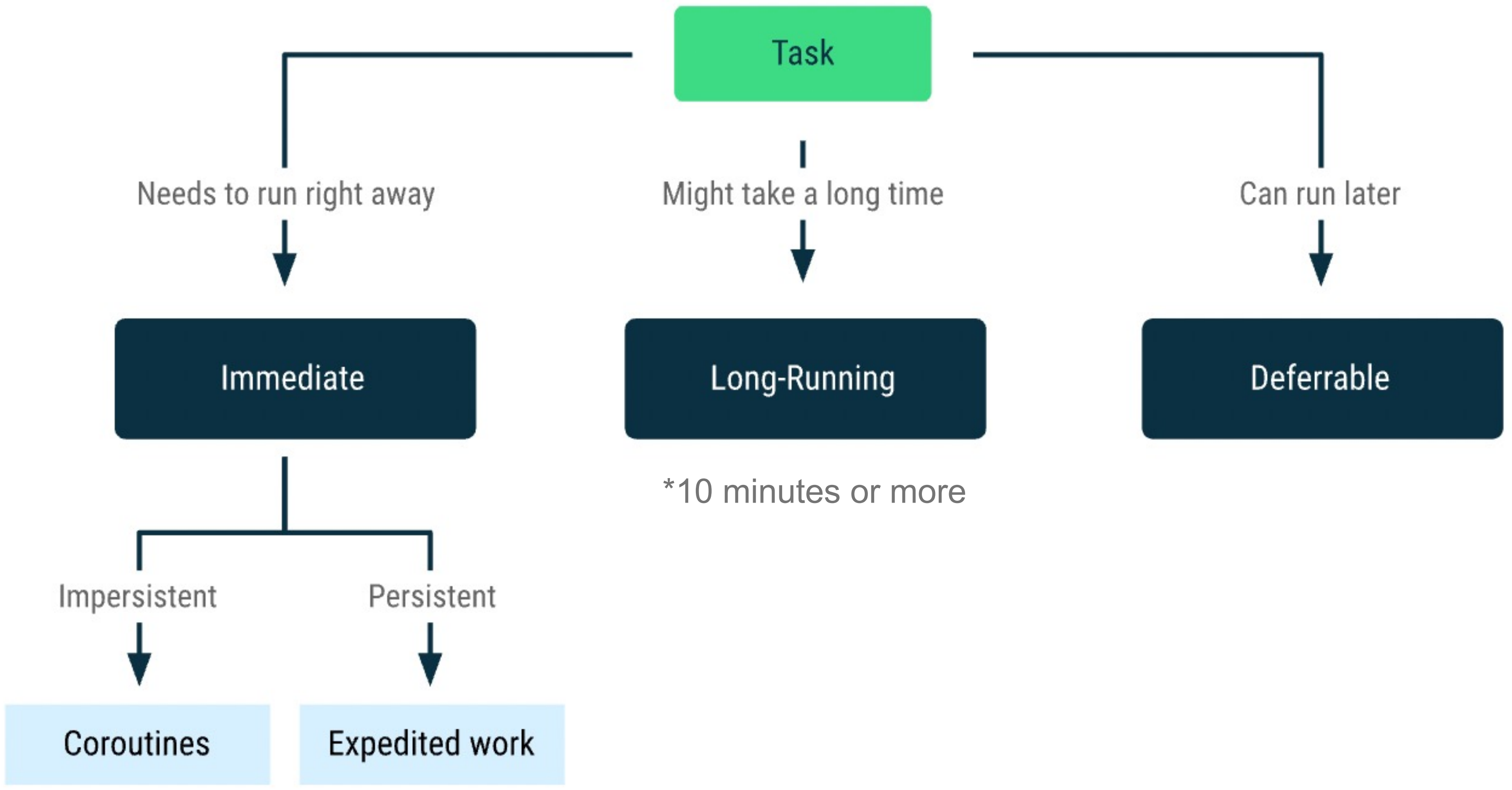


Figure 1: Types of background work.

Types of Background Work

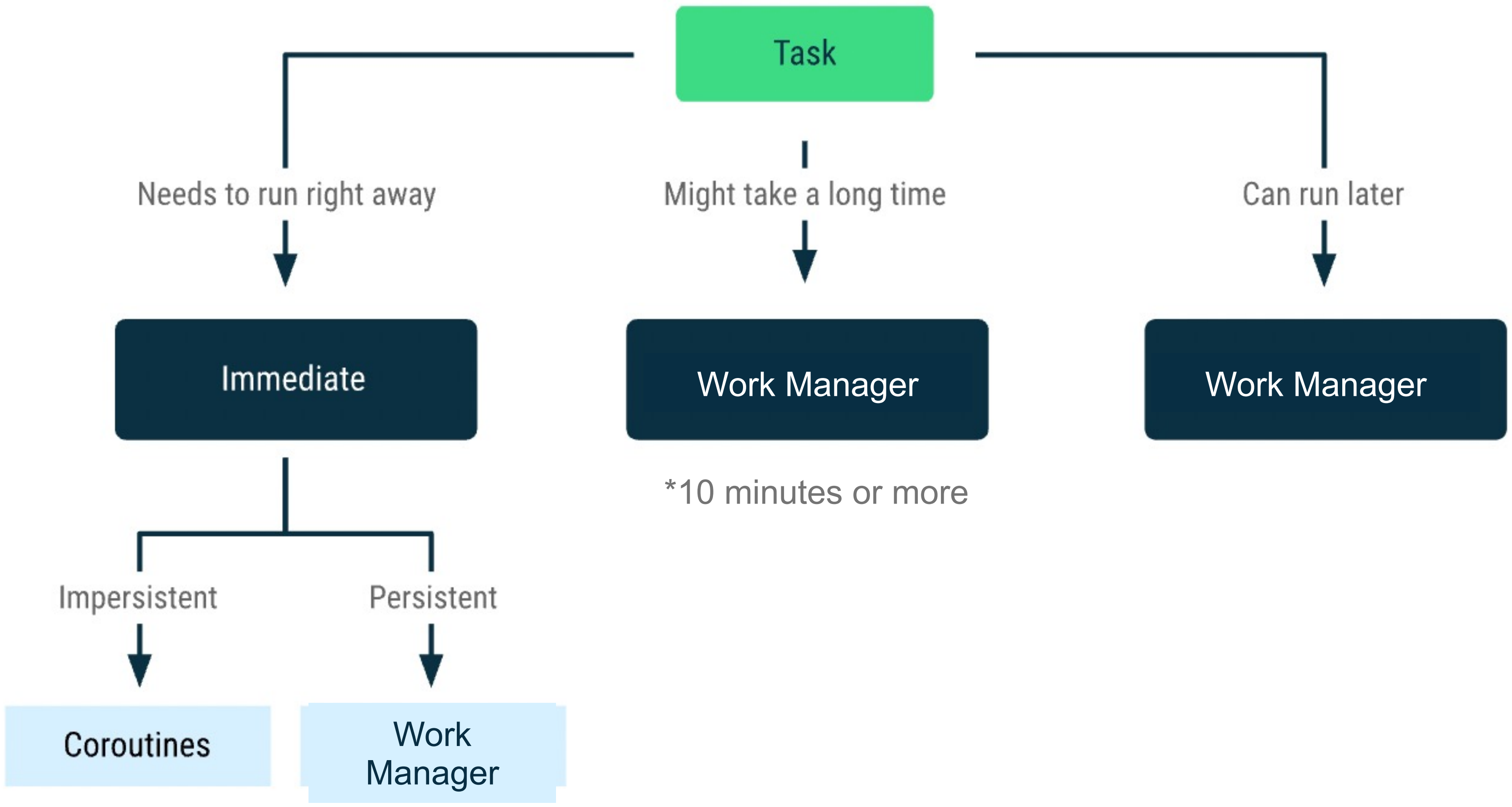


Figure 1: Types of background work.

Types Of Work

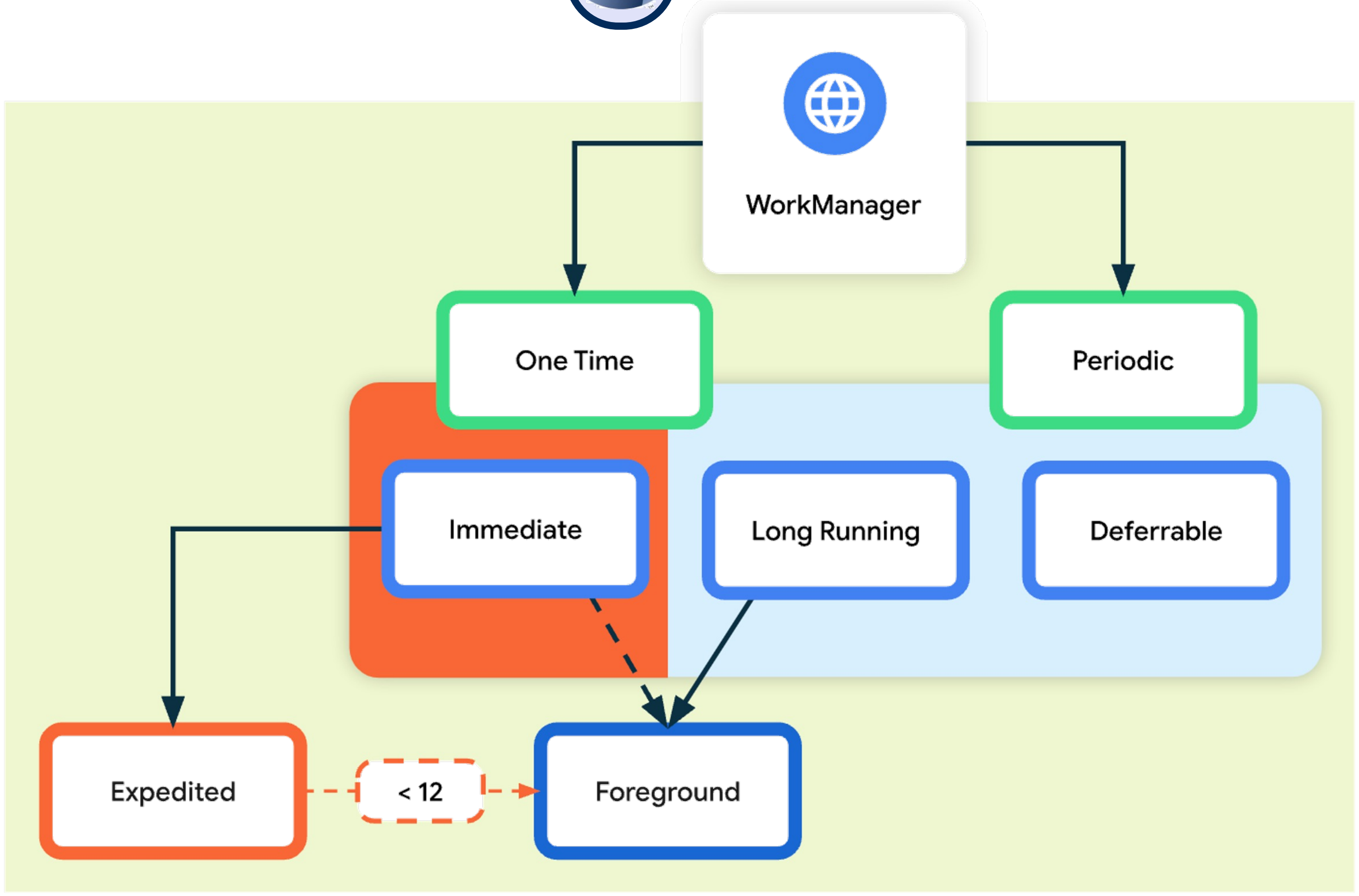


Figure 1: Types of persistent work.

On Tap For Today

- A Background Thread
- WorkManager
- Practice

WorkManager

- Executes WorkRequest on a separate thread

```
val workRequest = OneTimeWorkRequest
   .Builder(MyWorker::class.java)
   .build()

val workManager = WorkManager.getInstance(context)
workManager.enqueue( workRequest )
```

Android Design Patterns

- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State, Flow
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph, WorkRequest
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model
  8. Façade – DAO, Repository

Worker

- Create a Worker to actually do the work

```
class MyWorker(context: Content, workerParams: WorkerParameters)
        : Worker(context, workerParams) {

  override fun doWork(): Result {
    // do your task


    // if it succeeds
      return Result.success()
    // if it fails
      return Result.failure()
  }
}
```

Be Good Programmers

- Before doing anything you need to…

- Make sure you are connected to the internet
  - check connectivity and handle situation cleanly

Permissions

- Add to manifest
  - `<uses-permission android:name="…">`
- Must request permission to access network
  - `android.permission.INTERNET`
- Must request permission to check network state
  - `android.permission.ACCESS_NETWORK_STATE`

Protection Levels: Normal

- No great risk to privacy or security - user probably won't care.
- Still need to request the permission in the manifest, but system automatically grants (user not prompted).

ACCESS_NETWORK_STATE

ACCESS_WIFI_STATE

BLUETOOTH

CHANGE_NETWORK_STATE

CHANGE_WIFI_STATE

DISABLE_KEYGUARD

EXPAND_STATUS_BAR

GET_PACKAGE_SIZE

INSTALL_SHORTCUT

INTERNET

:

Now What?

- Do whatever it is you need the network for

- Make URL request

```
// java.net.URL is a blocking call
val websiteContentString = URL("http://...").readText()
// now contains contents at the web address
```

Worker Input

- Can provide inputData to the Worker

```
// when making request
val inData = workDataOf( "inKey" to value )
val workRequest = OneTimeWorkRequest
  .Builder(MyWorker::class.java)
  .setInputData(inData)
  .build()

// when doing work
override fun doWork(): Result {
  val inDataValue = inputData.get*("inKey")
  ...
}
```

Worker Input

- Encapsulate on Worker

```
// when making request
val inData = MyWorker.setInputData(value)
val workRequest = // setup from prior slide


// inside MyWorker
class MyWorker : ... {
  companion object {
    private const val INPUT_KEY = "inKey"
    fun setInputData(value: *) = workDataOf( INPUT_KEY to value )
  }
  override fun doWork() : Result {
    val inDataValue = inputData.get*(INPUT_KEY)
    ...
  }
}
```

Observing Worker State

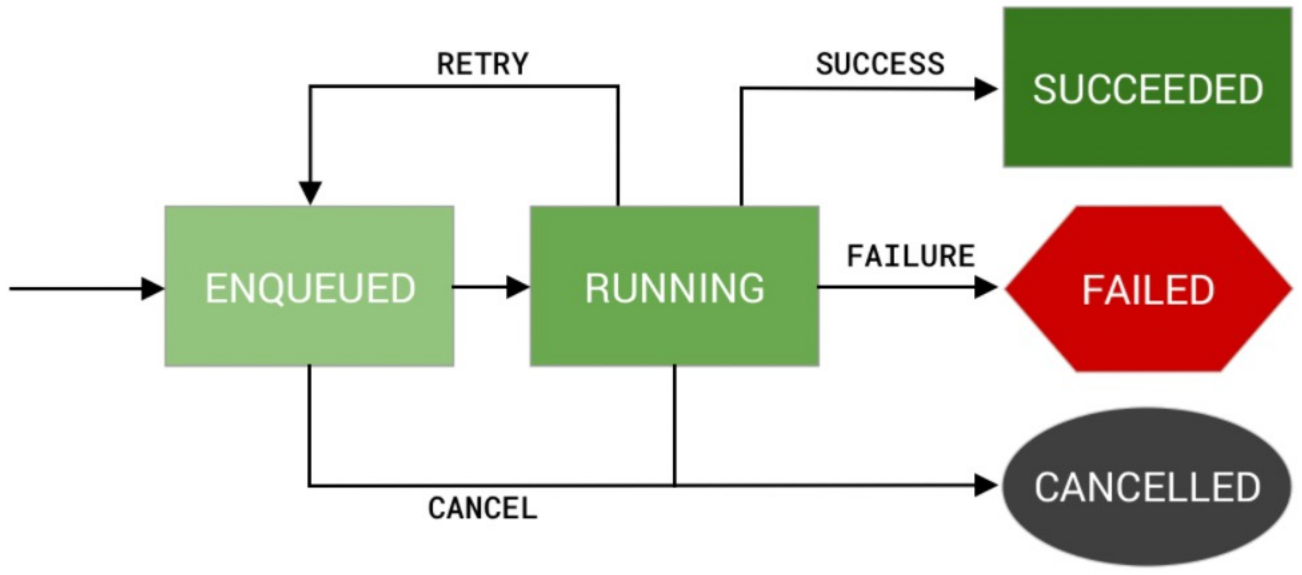


Figure 1. State diagram for one-time work.

Observing Worker State

- Use LiveData to track state changes

```
val workRequest = // setup from prior prior slide
workManager.enqueue(workRequest)
val workInfoState = workManager
        .getWorkInfoByIdLiveData(workRequest.id)
        .observeAsState()
workInfoState.value?.let { workInfo ->
  when(workInfo.state) {
      WorkInfo.State.RUNNING ->   // running
      WorkInfo.State.SUCCEEDED -> // done
      WorkInfo.State.CANCELLED -> // cancelled
  }
}
```

Android Design Patterns

- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State, Flow, LiveData
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph, WorkRequest
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model
  8. Façade – DAO, Repository

Worker Output

- Return with Result

```
// when doing work
override fun doWork(): Result {
  ...
  val outData = workDataOf( "key" to value )
  return Result.success(outData)
}

// when succeeded in observer
if(workInfo.state == WorkInfo.State.SUCCEEDED) {
  val outData = workInfo.outputData
  val outValue = outData.get*("key")
}
```

Worker Output

- Encapsulate on Worker again

```
class MyWorker {
  companion object {
    private const val OUTPUT_KEY = "outKey"
    fun getOutputData(outputData: Data) = outputData.get*(OUTPUT_KEY)
  }
  override fun doWork(): Result {
    ...
    val outData = workDataOf( OUTPUT_KEY to value )
    return Result.success(outData)
  }
}


// when succeeded in observer
if(workInfo.state == WorkInfo.State.SUCCEEDED) {
  val outValue = MyWorker.getOutputData( workInfo.outputData )
  ...
}
```

Observing Worker Progress

- While doing the work, set the current progress

```
class MyWorker {
  companion object {
    private const val PROGRESS_KEY = "progKey"
    fun getProgress(progressData: Data) = progressData.getInt(PROGRESS_KEY, 0)
  }
  override fun doWork(): Result {
    ...
    val updateData = workDataOf( PROGRESS_KEY to intValue )
    setProgress(updateData)
    ...
  }
}


// when running in observer
if(workInfo.state == WorkInfo.State.RUNNING) {
  val progress = MyWorker.getProgress( workInfo.progress )
  // do something with the value, like update a progress bar or something
}
```

Full Worker Shell

```
class MyWorker(context: Content, workerParams: WorkerParameters)
        : Worker(context, workerParams) {
  companion object {
    private const val INPUT_KEY = "inKey"
    fun setInputData(value: *) = workDataOf( INPUT_KEY to value )

    private const val PROGRESS_KEY = "progKey"
    fun getProgress(progressData: Data) = progressData.getInt(PROGRESS_KEY, 0)

    private const val OUTPUT_KEY = "outKey"
    fun getOutputData(outputData: Data) = outputData.get*(OUTPUT_KEY)
  }
  override fun doWork(): Result {
    val inDataValue = inputData.get*(INPUT_KEY)
    while(/*running*/) {
      val updateData = workDataOf( PROGRESS_KEY to progressIntegerValue )
      setProgress(updateData)
      // do your task & update progress integer value
    }
    // if it succeeds
    val outData = workDataOf( OUTPUT_KEY to outputValue )
    return Result.success(outData)
    // if it fails
    return Result.failure()
  }
}
```

Full Work Request Shell

```
val inData = MyWorker.setInputData(value)

val workRequest = OneTimeWorkRequest
   .Builder(MyWorker::class.java)
   .setInputData(inData)
   .build()

val workManager = WorkManager.getInstance(context)
workManager.enqueue( workRequest )
```

Full Observer Shell

```
val workRequest = // setup from prior slide

workManager.enqueue(workRequest)

val workInfoState = workManager
        .getWorkInfoByIdLiveData(workRequest.id)
        .observeAsState()

workInfoState.value?.let { workInfo ->
  when(workInfo.state) {
      WorkInfo.State.RUNNING -> {
        // running
        val progress = MyWorker.getProgress( workInfo.progress )
      }
      WorkInfo.State.SUCCEEDED -> {
        // done
        val outValue = MyWorker.getOutputData( workInfo.outputData )
      }
      WorkInfo.State.CANCELLED -> { // cancelled }
  }
}
```

More Complex Work

- Can chain individual work together into a sequence with dependencies
- Can set constraints on what is needed for work to run (WiFi, Battery Level, etc)
- Can replace existing work if already running

Android Design Patterns

- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State, Flow, LiveData
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph, WorkRequest, Constraints
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model
  8. Façade – DAO, Repository

On Tap For Today

- A Background Thread
- WorkManager
- Practice

To Do For Next Time

- Exam 1 due Fri Mar 03 – tonight
- Lab06 due Tue Mar 07
- Lab07 due Fri Mar 10
- Alpha Release due Mon Mar 13 – have NavGraph in place
- A2 due Tue Mar 14
- Lab08 due Fri Mar 17
- Alpha Feedback due Fri Mar 17
- Spring Break !!!