LESSON 04 ANDROID

AARHUS
UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER
ENGINEERING

25 OCTOBER 2022

MICHEL VEDEL HOWARD
ASSISTANT PROFESSOR

CONTENT

1. Activity
2. Intent
3. Activity Life Cycle
4. Multiple Activities
5. Task and back stack
6. Multiple Tasks
7. More Intent

ACTIVITY

Activity is a top-level component of an app. It can be the thought of as a screen.

Apps can have multiple Activities. In JetPack

Composables are the nodes under the activity

An Activity is configured in the AndroidManifest.xml file

located in app/src/main/

```
<application
    android:allowBackup="true"
    android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="DrawerApp"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Theme.DrawerApp"
    tools:targetApi="31">
    <activity
        android:name=".MainActivity"
        android:exported="true"
        android:theme="@style/Theme.DrawerApp">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
```

MANIFEST ACTIVITY

```
<activity
    android:name=".MainActivity"
    android:exported="true"
    android:theme="@style/Theme.DrawerApp">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Class representation

If true, the activity can be activated from extern apps

Reference to style resource

A filter that restricts other apps/OS interaction with this activity

INTENT INTRO

Three of the core components of an application
— **activities**, services, and broadcast receivers
are activated through messages, called intents.
The intent filter(s) specify matching rules for an activity on these intents.

This activity admits itself to be listed in the top-level application launcher
and is the activity created when launched. This combination of action and category is used for making the app appear as a launchable app

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

THE LIFE OF ACTIVITY

A composable App contains at least one Activity
and the life of an activity is schematically depicted as

The different (On)methods described in the figure can be overridden to hook into the activity lifecycle.

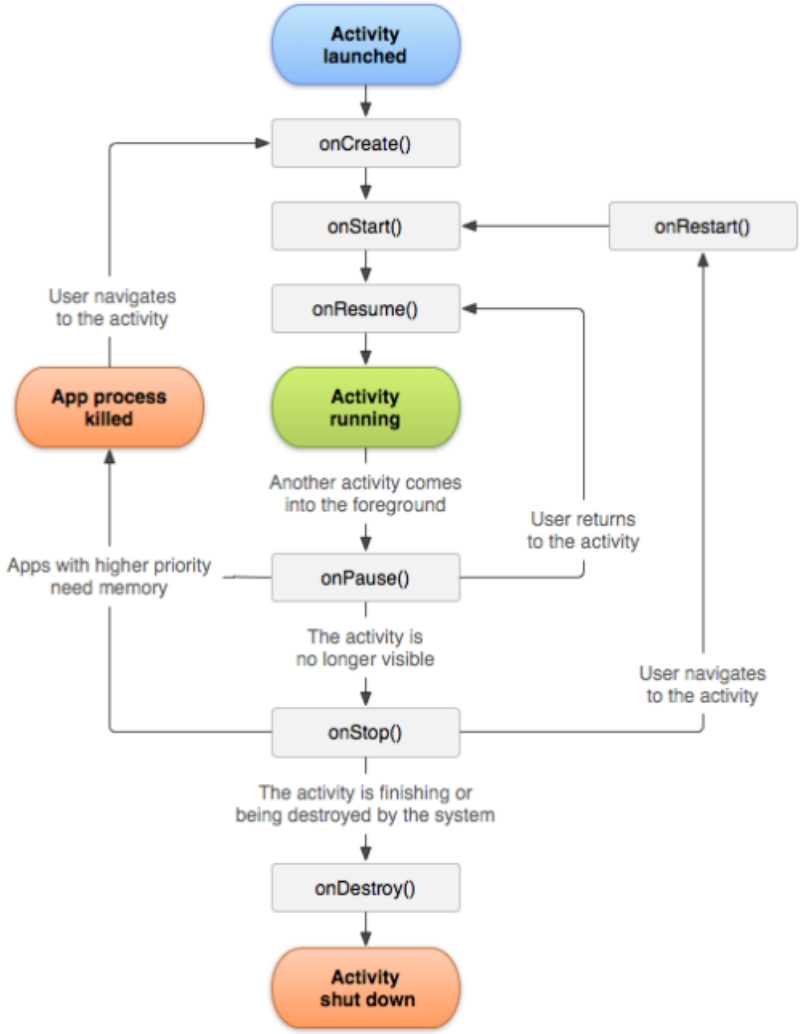


Figure 1. A simplified illustration of the activity lifecycle.

LIFE OF PI EXAMPLE

We look at an example with two Activities. The AndroidManifest.xml looks like

```
<activity
    android:name=".activities.pi.PiActivity"
    android:exported="true"
    android:theme="@style/Theme.LifeOfPi">
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>

</activity>
```

```
<activity
    android:name=".activities.pi.TigerActivity"
    android:exported="true"
    android:theme="@style/Theme.LifeOfPi" />
```

PI

```
class PiActivity : LoggedActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            LifeOfPiTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Column() { this: ColumnScope
                        Text( text: "PI Activity")
                        Button(onClick = {
                            Log.v(this@PiActivity::class.simpleName,  msg: "Go to Tiger")
                            val intent = Intent( packageContext: this@PiActivity, TigerActivity::class.java)
                            intent.putExtra( name: "test", value: 5)
                            startActivity(intent)
                        }) { this: RowScope
                            Text( text: "Go to Tiger")
                        }
                    }
                }
            }
        }
    }
}
```

TIGER

```
class TigerActivity : LoggedActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            LifeOfPiTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Column() { this: ColumnScope
                        Text( text: "Tiger received:${intent.getIntExtra( name: "test",  defaultValue: 0)} from Pi")
                        Button(onClick = {
                            Log.v(this::class.simpleName,  msg: "Back to Pi")
                            finish()
                        }) { this: RowScope
                            Text( text: "Back to Pi")
                        }
                    }
                }
            }
        }
    }
}
```

LIFE OF PI

So, two activities one with a launcher which is PiActivity

Now for logging lifecycle hook calls

Defined a LoggedActivity that logs in all hooks

And now look at the life

```
open class LoggedActivity : ComponentActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        Log.v(this::class.simpleName,  msg: "CREATING")
    }

    override fun onStart() {
        super.onStart()
        Log.v(this::class.simpleName,  msg: "STARTING")
    }

    override fun onResume() {
        super.onResume()
        Log.v(this::class.simpleName,  msg: "RESUMING")
    }

    override fun onRestart() {
```

LIFE OF PI

Starting the app, the log shows:

```
PiActivity              dk.howards.lifeofpi              V  CREATING
PiActivity              dk.howards.lifeofpi              V  STARTING
PiActivity              dk.howards.lifeofpi              V  RESUMING
```

Which is consistent with the lifecycle diagram

Now we jump ->

LIFE OF PI

```
PiActivity          dk.howards.lifeofpi    V  Go to Tiger
PiActivity          dk.howards.lifeofpi    V  PAUSEING
ActivityThread      dk.howards.lifeofpi    W  handleWindo
TigerActivity       dk.howards.lifeofpi    V  CREATING
TigerActivity       dk.howards.lifeofpi    V  STARTING
TigerActivity       dk.howards.lifeofpi    V  RESUMING
EGL_emulation       dk.howards.lifeofpi    D  eglMakeCurr
PiActivity          dk.howards.lifeofpi    V  STOPPING
```

We notice consistency Pi pauses and Tiger creates

and note that Pi goes into stop after tiger is created because Pi gets overpowered by Tiger.

Pi is no longer visible. Now wake up Pi and push back button ->

LIFE OF PI

```
ColumnScopeInstance     dk.howards.lifeofpi     V  Back to Pi
TigerActivity           dk.howards.lifeofpi     V  PAUSEING
PiActivity              dk.howards.lifeofpi     V  RESTARTING
PiActivity              dk.howards.lifeofpi     V  STARTING
PiActivity              dk.howards.lifeofpi     V  RESUMING
EGL_emulation           dk.howards.lifeofpi     D  eglMakeCurre
TigerActivity           dk.howards.lifeofpi     V  STOPPING
TigerActivity           dk.howards.lifeofpi     V  Destroying
```

The Tiger pauses and Pi being stopped restarts starts and resumes

The tiger stops and destroys!!

This is because of command **finish**

LIFE OF PI

Now go to Tiger again

```
PiActivity          dk.howards.lifeofpi   V  Go to Tiger
PiActivity          dk.howards.lifeofpi   V  PAUSEING
ActivityThread      dk.howards.lifeofpi   W  handleWindo
TigerActivity       dk.howards.lifeofpi   V  CREATING
TigerActivity       dk.howards.lifeofpi   V  STARTING
TigerActivity       dk.howards.lifeofpi   V  RESUMING
EGL_emulation       dk.howards.lifeofpi   D  eglMakeCurr
PiActivity          dk.howards.lifeofpi   V  STOPPING
```

TASK AND BACK STACK

The Task is started when a program is started from say the home screen

The MAIN activity is pushed on top of the back-stack

When the current activity starts a new activity, it is pushed on the stack and gains focus.

The previous activity might persist on the stack but according to life cycle diagram go into say stopped state. But it remembers its internal state.

When wanting to go back the stack is manipulated at your desire. By maybe popping or swapping.

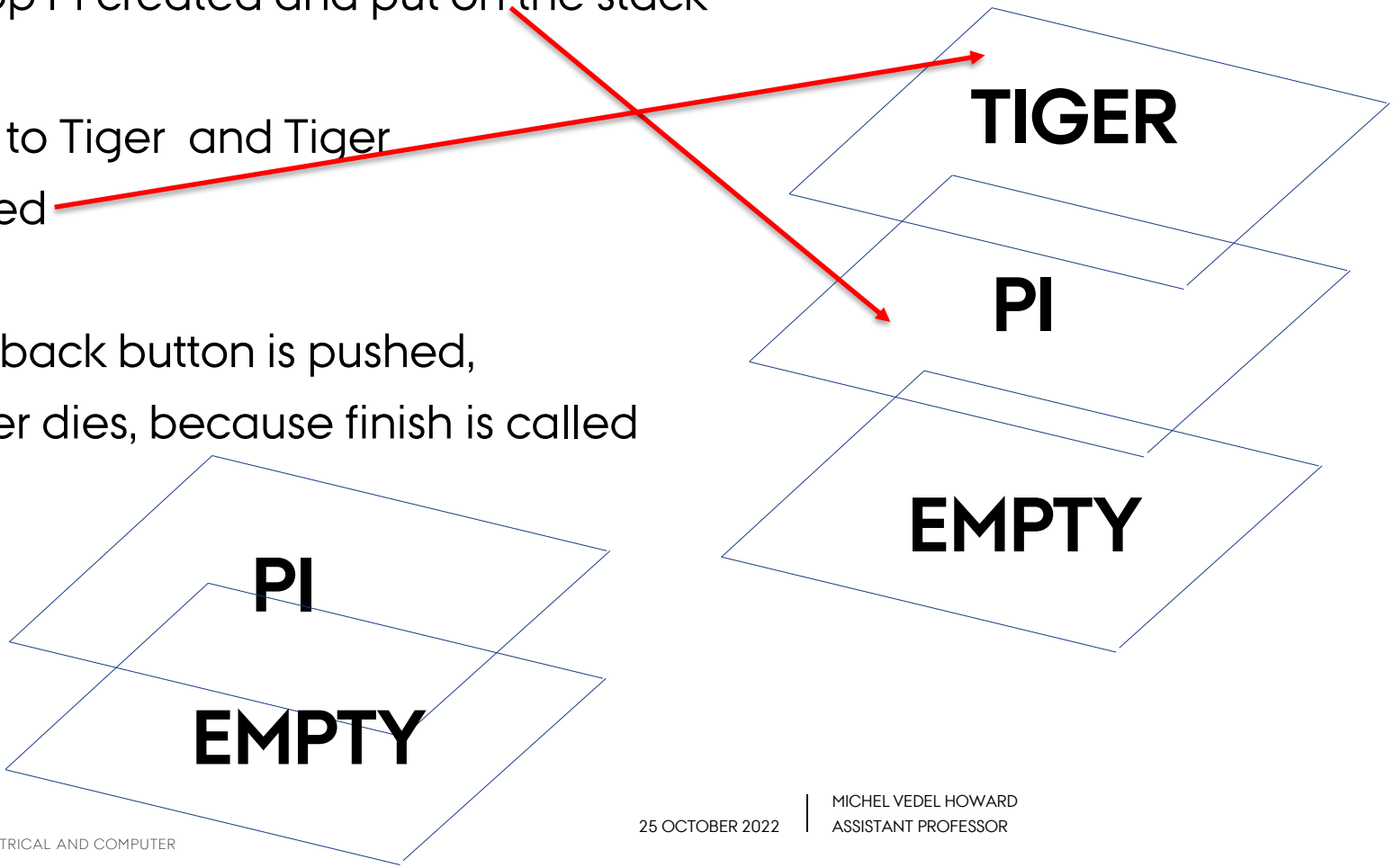
The typical thing to do though is to pop the activity and recreate if needed again.

TASK AND ANDROID ACTIVITY BACK STACK

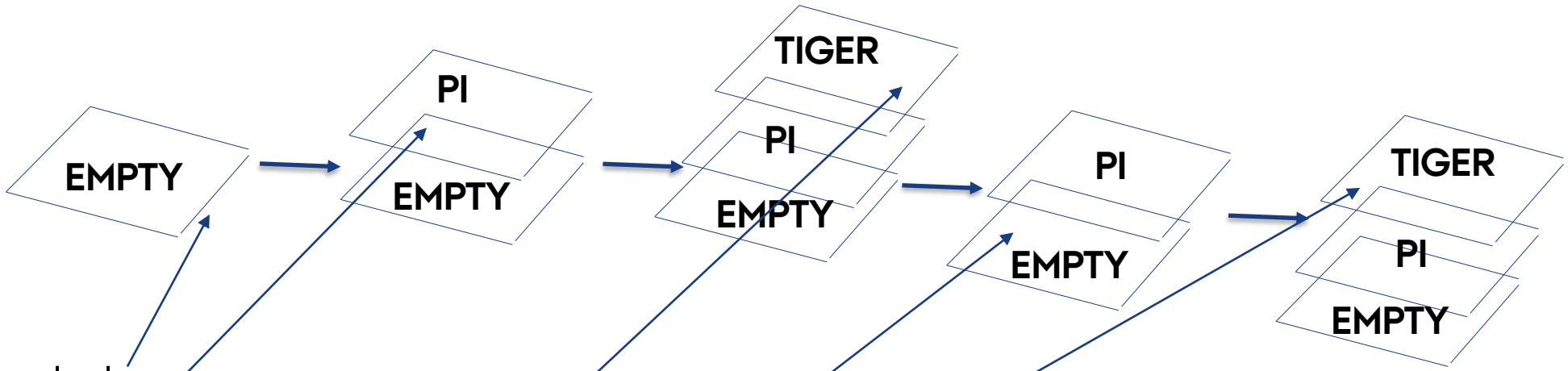
Start App Pi created and put on the stack

The Go to Tiger and Tiger is created

But the back button is pushed, and tiger dies, because finish is called



ANDROID ACTIVITY BACK STACK



App start

Pi created and pushed

Jump: Tiger created and pushed

Back : Tiger finished and popped

Jump: Tiger created and pushed

MANIPULATING THE STACK

Could we do this differently could we swap PI and Tiger instead of creating and killing tiger

If we look at the documentation, we find that a flag can be added to the intent

```
intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
```

MANIPULATING THE STACK

On Pi

```
Button(onClick = {
    Log.v(this@PiActivity::class.simpleName, msg: "Go to Tiger")
    val intent = Intent( packageContext: this@PiActivity, TigerActivity::class.java)
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    startActivity(intent)
}) {  this: RowScope
    Text( text: "Go to Tiger")
}
```

On Tiger

```
Button(onClick = {
    Log.v(this::class.simpleName,  msg: "Back to Pi")
    val intent = Intent( packageContext: this@TigerActivity, PiActivity::class.java)
    intent.addFlags(Intent.FLAG_ACTIVITY_REORDER_TO_FRONT)
    intent.putExtra( name: "SWAPPING",  value: 1)
    //finish()
    startActivity(intent)
}) {  this: RowScope
    Text( text: "Back to Pi")
}
```

LOG

```
PiActivity              dk.howards.lifeofpi          V  CREATING
PiActivity              dk.howards.lifeofpi          V  STARTING
PiActivity              dk.howards.lifeofpi          V  RESUMING
```

```
PiActivity              dk.howards.lifeofpi          V  Go to Tiger
PiActivity              dk.howards.lifeofpi          V  PAUSEING
ActivityThread          dk.howards.lifeofpi          W  handleWindow\
TigerActivity           dk.howards.lifeofpi          V  CREATING
TigerActivity           dk.howards.lifeofpi          V  STARTING
TigerActivity           dk.howards.lifeofpi          V  RESUMING
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurrer
PiActivity              dk.howards.lifeofpi          V  STOPPING
```

```
ColumnScopeInstance     dk.howards.lifeofpi          V  Back to Pi
TigerActivity           dk.howards.lifeofpi          V  PAUSEING
PiActivity              dk.howards.lifeofpi          V  RESTARTING
PiActivity              dk.howards.lifeofpi          V  STARTING
PiActivity              dk.howards.lifeofpi          V  OnNewIntent null
PiActivity              dk.howards.lifeofpi          V  RESUMING
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurrent: 0xf676ce00: ver 2 0 (tinfo 0xf67a2ab0)
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurrent: 0xf676ce00: ver 2 0 (tinfo 0xf67a2ab0)
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurrent: 0xf676ce00: ver 2 0 (tinfo 0xf67a2ab0)
TigerActivity           dk.howards.lifeofpi          V  STOPPING
```

```
PiActivity              dk.howards.lifeofpi          V  Go to Tiger
PiActivity              dk.howards.lifeofpi          V  PAUSEING
TigerActivity           dk.howards.lifeofpi          V  RESTARTING
TigerActivity           dk.howards.lifeofpi          V  STARTING
TigerActivity           dk.howards.lifeofpi          V  OnNewIntent
TigerActivity           dk.howards.lifeofpi          V  RESUMING
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurre
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurre
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurre
chatty                  dk.howards.lifeofpi          I  uid=10136(dk
EGL_emulation           dk.howards.lifeofpi          D  eglMakeCurre
PiActivity              dk.howards.lifeofpi          V  STOPPING
```

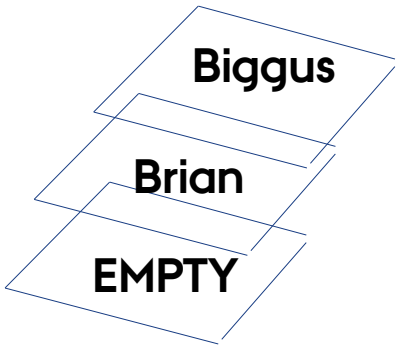
MORE THAN ONE TASK

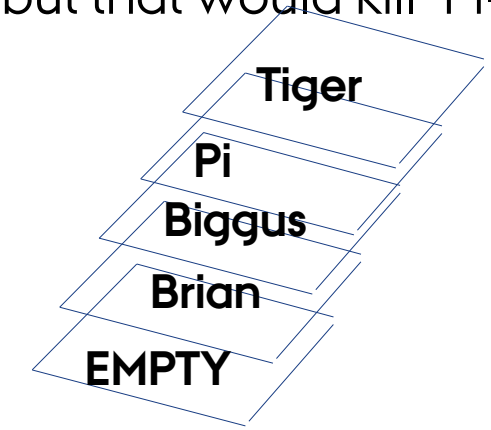
We could be in the position where several tasks could be interesting

Let us say that I have resource-demanding set of Activities Brian->BiggusDickus

And I have another set of activities Pi->Tiger

Now imagine that Pi and tiger is pushed on top. If I would like to bring back Brian –>Biggus I could clear the stack down to that point but that would kill Pi->Tiger and that might be undesired

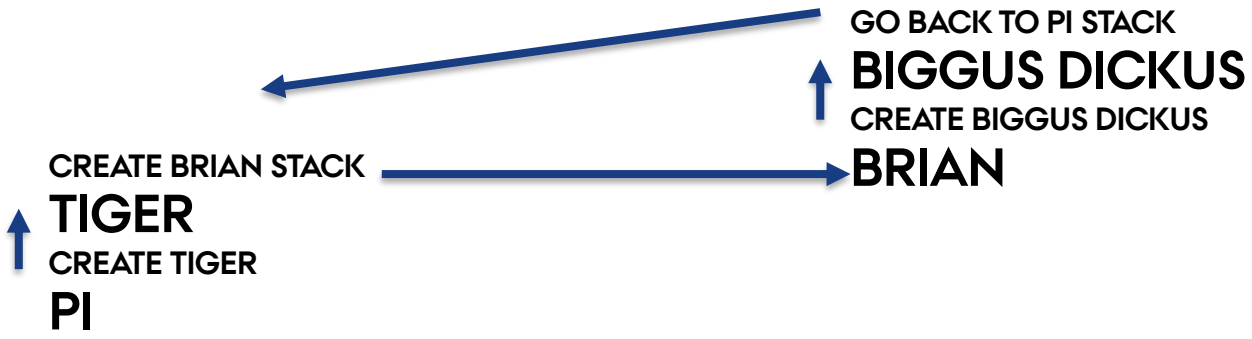




TWO TASKS

Instead, I could arrange for two tasks, that is two stacks at once, and jump between them

Without destroying the stacks



JUMPING STACKS

This can be obtained in different ways. First one should associate the activity with a new Task. In this case named LifeOfBrian.task. Second Flag NEW_TASK in intent

```
<activity
    android:name=".activities.brian.BrianActivity"
    android:exported="true"
    android:theme="@style/Theme.LifeOfPi"
    android:taskAffinity="LifeOgBrian.task"
    />
```

```
Button(onClick = {
    Log.v(this@TigerActivity::class.simpleName, msg: "Create Important LifeOfBrian task")
    val intent = Intent( packageContext: this@TigerActivity, BrianActivity::class.java)
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
    startActivity(intent)
}) { this: RowScope
    Text( text: "Create Important Brian task")
}
```

JUMPING STACKS

And to jump back to Pi from BrianActivity->BiggusDickus we simply use an intent with new Task flag to jump back to the MAIN task, which if task and Activity is present will reuse activity

```
Button(onClick = {
    Log.v(this::class.simpleName,  msg: "Return to Main")
    val intent = Intent( packageContext: this@BiggusDickusActivity, PiActivity::class.java)
    intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
    intent.action = Intent.ACTION_MAIN
    intent.addCategory(Intent.CATEGORY_LAUNCHER)
    startActivity(intent)
}) { this: RowScope
    Text( text: "GO TO PI MAIN TASK")
}
```

And to jump back to LifeOfBrian from Tiger we simply reuse the same method we used to create the LifeOfBrian.Task because the task is still running

INTENT IN DEPTH

There are two primary forms of intents you will use.

- **Explicit Intents** have specified a component (via `setComponent(ComponentName)` or `setClass(Context, Class)`), which provides the exact class to be run. Often these will not include any other information, simply being a way for an application to launch various internal activities it has as the user interacts with the application.
- **Implicit Intents** have not specified a component; instead, they must include enough information for the system to determine which of the available components is best to run for that intent.

As we can see our intents so far have been explicit!

This constructor of intent takes as input a kind of from-to information, where the first parameter is where you come from and second the activity class you want to go to

```
val intent = Intent( packageContext: this@PiActivity, TigerActivity::class.java)
```

And further data can be set such as action, category flags and associated data.

EXPLICIT INTENTS

Making an explicit **intent** may involve sending data to the new Activity

Let us look at examples which implement this feature .

We look at three cases

1. Send a “primitive” piece of data Int, Long, String….
2. Send a self-made object using Serializable technique
3. Send a self-made object using Parcelable technique

Then we can use the *putExtra* method to place a Person object on the intent.

The Person object must be of type Serializable, so it most implement interface **Serializable** or it can implement **Parcelable**

We can also use the **data** property of intent that is of type **Uri**

Or we can use *putExtras* with “simple” types

PRIMITIVE DATA

EXAMPLE/PARCELABLE

We consider in this case also strings to be primitive

SEND

```
Button(onClick = {
    val intent = Intent( packageContext: this@SendingActivity, ReceivingActivity::class.java)
    intent.putExtra( name: "Message",  value: "Hello")
    startActivity(intent)
}) { this: RowScope
    Text( text: "Sending a string hello to Receiving Activity")
}
```

RECEIVE

```
Row { this: RowScope
    Text( text: "Message")
    Text( text: intent?.getStringExtra( name: "Message") ?: "")
}
```

SENDING OBJECT CAT BY SERIALIZABLE

EXAMPLE/PARCELABLE

The serializable data class

```
data class Cat(val tagNumber: Long, val name: String) : Serializable
```

SEND

```
Button(onClick = {
    val intent = Intent( packageContext: this@SendingActivity, ReceivingActivity::class.java)
    intent.putExtra( name: "Cat", Cat( tagNumber: 1,  name: "Stray Cat"))
    startActivity(intent)
}) { this: RowScope
    Text( text: "Sending serializable Cat to Receiving Activity")
}
```

RECEIVE

```
var cat: Cat? = null
if (intent.extras?.containsKey("Cat") == true) {
    cat = intent.getSerializableExtra(
        name: "Cat",
        Cat::class.java
    )
}
```

```
Row { this: RowScope
    Text( text: "TagNumber")
    Text( text: cat?.tagNumber?.toString() ?: "")
}
Row { this: RowScope
    Text( text: "Name")
    Text( text: cat?.name ?: "")
}
```

SENDING OBJECT PERSON BY PARCELABLE

EXAMPLE/PARCELABLE

First, a plugin needs to be installed in the module Gradle file to use this feature

```
plugins { this: PluginDependenciesSpecScope
    id("com.android.application")
    id("org.jetbrains.kotlin.android")
    id("kotlin-parcelize")
}
```

The class we want to send object of

```
@Parcelize
data class Person(val cpr: Long, val name: String) : Parcelable
```

SENDING OBJECT PERSON BY PARCELABLE

EXAMPLE/PARCELABLE

SEND

```
Button(onClick = {
    val intent = Intent( packageContext: this@SendingActivity, ReceivingActivity::class.java)
    intent.putExtra( name: "Person", Person( cpr: 1,  name: "John Doe"))
    startActivity(intent)
}) { this: RowScope
    Text( text: "Sending parcelable person to Receiving Activity")
}
```

RECEIVE

```
var person: Person? = null
if (intent.extras?.containsKey("Person") == true) {
    person = intent.getParcelableExtra(
        name: "Person",
        Person::class.java
    )
}
```

```
Row { this: RowScope
    Text( text: "Cpr")
    Text( text: person?.cpr?.toString() ?: "")
}
Row { this: RowScope
    Text( text: "Name")
    Text( text: person?.name ?: "")
}
```

There is one snag though. The method getParceableExtra requires minSDK 33

There is another method that might work on lower SDK but that one is deprecated.

So, using parcelable requires min SDK 33 and you should have an emulator that matches

DIFFERENCE SERIALIZABLE AND PARCELABLE

Serializable relies on reflection and is more performance expensive

Parcelable is more efficient and is worth using at least for large objects.

REGISTER ACTIVITY RESULT CALLBACK

EXAMPLE/REGISTERFORRESULT

In order, for an activity to return data to the caller, a callback must be registered with the activity started. In constructor:

```
private val registerForResult =
    registerForActivityResult(ActivityResultContracts.StartActivityForResult()) { it: ActivityResult!
        Log.v(this::class.simpleName,  msg: "Call back received")
        Log.v(this::class.simpleName, it.resultCode.toString())
        Log.v(this::class.simpleName, it.data.toString())

    }
```

Launch intent

```
Button(onClick = {
    val intent = Intent( packageContext: this@StartActivity, RespondingActivity::class.java)
    registerForResult.launch(intent)
}) { this: RowScope
    Text( text: "Go to Other Activity")
}
```

RETURN RESULT

EXAMPLE/REGISTERFORRESULT

```
Button(onClick = {
    val intent = Intent( packageContext: this@RespondingActivity, StartActivity::class.java)
    intent.putExtra( name: "RETURN-VALUE",  value: 1234)
    setResult(RESULT_OK, intent)
    finish()
}) { this: RowScope
    Text( text: "Back")
}
```

IMPLICIT INTENTS

Implicit intents do not have a specific target but are resolved by the OS in various ways

This is what is called Intent resolution

The intent can be used for:

1. Starting an Activity
2. Starting a Service (a service for decryption say)
3. Broadcast for action (example battery power broadcasts)(

Here is where the intent filters really come into play.

The filtering is based on the **action**, (mime)**type**, and **category**

A query is made to the **PackageManager** to resolve the activity.

THE FILTERING

One or more activities are resolved if:

Action must be in filter and match

Data Type(mime type) must be in filter

Category(ies) must all be in filter

If several Apps have filters that match the “broadcasted” intent a chooser is typically presented to the user

MIME TYPES

```
image/jpeg
audio/mpeg4-generic
text/html
audio/mpeg
audio/aac
audio/wav
audio/ogg
audio/midi
audio/x-ms-wma
video/mp4
video/x-msvideo
video/x-ms-wmv
image/png
image/jpeg
image/gif
.xml ->text/xml
.txt -> text/plain
.cfg -> text/plain
.csv -> text/plain
.conf -> text/plain
.rc -> text/plain
.htm -> text/html
.html -> text/html
.pdf -> application/pdf
.apk -> application/vnd.android.package-archive
```

IMPLICIT INTENT

EXAMPLES/IMPLICITINTENT

The App that asks the OS for an activity, that can handle a specific intent is the ImplicitIntent app

```
Button(onClick = {
    Log.v(this::class.simpleName,  msg: "TRYING TO LAUNCH INTENT")
    val intent = Intent(Intent.ACTION_EDIT)
    intent.type = "text/plain"
    intent.putExtra(Intent.EXTRA_TEXT,  value: "For Editor")
    val info = intent.resolveActivity(packageManager)
    Log.v(this::class.simpleName,  msg: info?.toString() ?: "Nothing")
    startActivity(intent)
```

We seek an app that can handle action EDIT and mime type text/plain and we deliver data *"For Editor".*

Trying to launch this intent gives no result. So, we will construct an app that can work as a simple editor with the right intent filter.

THE EDITOR APP

EXAMPLES/EDITOR

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContent {
            EditorTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Greeting( name: intent.getStringExtra(Intent.EXTRA_TEXT)?:"")
                }
            }
        }
    }
}
```

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
<intent-filter>
    <action android:name="android.intent.action.EDIT" />
    <data android:mimeType="text/plain" />
    <category android:name="android.intent.category.DEFAULT" />
</intent-filter>
```

THE MANIFEST

We want this app to be able to handle action *EDIT* (which we will interpret as an editor)

We want a match on no category, so we use *DEFAULT* category

The datatype we expect to handle is *text/plain*

BUT

We also want The app to be launchable which results in two intent filters

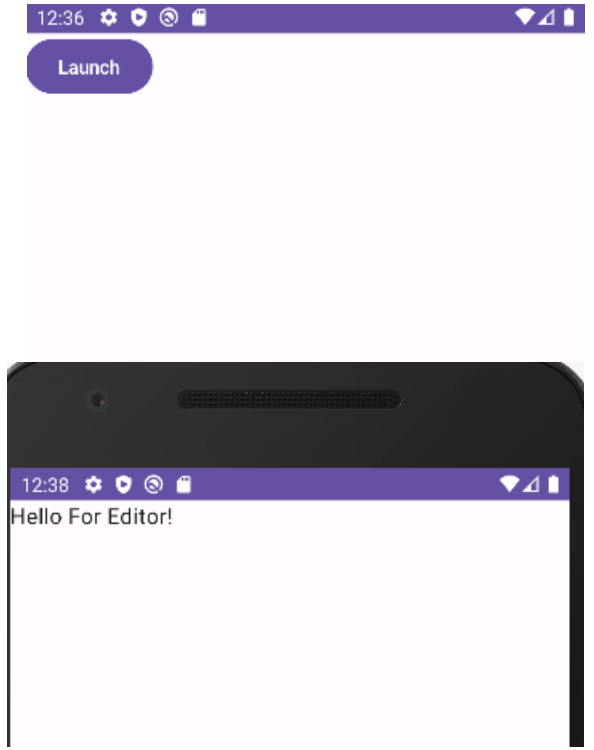
IMPLICIT INTENT EXAMPLE

Now we deploy The App that handles the EDIT action and close the app

Then we deploy the ImplicitIntent app

And when we push the launch button the EditorApp is

Automatically launched



AARHUS
UNIVERSITY