Module 5

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 5 Study Guide and Deliverables

Topics:	• Broadcast Receivers • SAsynchronous Processing • Services
Readings:	• Online lectures • Notes and references cited in lectures • Chapters 17-19 from the "Head First" textbook (or corresponding chapters related to topics covered in this module in other textbooks)
Discussions:	• Discussion 5 ◦ Initial post due **Tuesday, August 9 at 6:00am ET** ◦ Respond to threads posted by others due **Thursday, August 11 at 6:00am ET**
Labs:	• Lab 5 due **Tuesday, August 9 at 6:00am ET**
Assignments:	• Assignment 5 due **Tuesday, August 9 at 6:00am ET**
Assessments:	• Quiz 5 due **Tuesday, August 9 at 6:00am ET**
Live Classrooms:	• **Monday, August 8 from 6:00-8:00pm ET**

Learning Outcomes

By the end of this module, you will be able to:

- Describe how to send broadcast intents and implement broadcast receivers.
- Compare explicit intents and implicit intents.
- Describe the intent filters.
- Compare processes and threads.
- Describe how to create threads and how to communicate between threads using handlers.
- Describe how to use Kotlin Coroutines.
- Describe how to implement various types of services.
- Describe how to use WorkManager.
- Develop Android applications that may involve above features

Introduction

So far, we have discussed two different Android components: activities and content providers. We also introduced intents briefly. In this module, we will discuss two other Android components in detail: broadcast receivers and services. We will also discuss intents in more detail. In addition, we will discuss how to create a multi-threaded application for asynchronous processing in Android. Since services are commonly used for long run operations in the background, they are usually executed in separate threads.

Topic 1: Broadcast Receivers

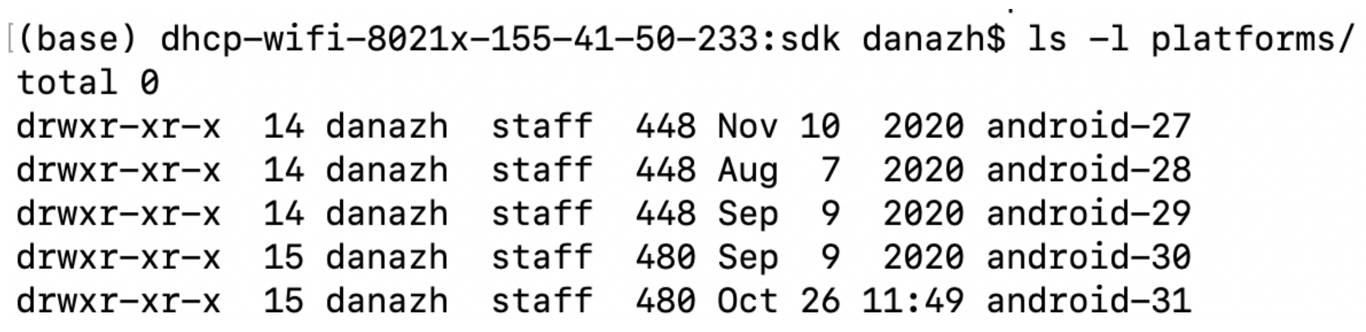
Broadcast Receivers

In Module 2, we discussed intents and ICC (Inter component communication). An intent is a message used to communicate between different components. When we need to switch from one activity to another activity, an intent is sent to the target activity. Sometimes, a component may want to communicate with multiple components, notifying all of them of some event. A **broadcast receiver** (**receiver**) is an Android component that can register for some system or application events. The events are described using intent actions. The sender broadcasts an intent with a specific action string specified and all registered **receivers** for that action are notified by the Android runtime once this event happens.

Broadcast Intents

Android applications can receive broadcast messages from the Android system or other Android apps. For example, the Android system automatically sends broadcast messages when the system boots up or when the battery is low. Applications can also send custom broadcasts to other applications. A broadcast message is wrapped in an Intent object with the action string specified and possibly some additional data (as a bundle). The action strings specify the types of events to notify. For example, when the airplane mode changes, the system broadcasts an intent with an action string “android.intent.action.AIRPLANE_MODE” and a boolean extra to indicate whether AIRPLANE_MODE is on or off. Here are some examples of broadcast actions: ACTION_SCREEN_ON, ACTION_BATTERY_LOW, ACTION_HEADSET_PLUG, BOOT_COMPLETED, CAMERA_BUTTON, etc.

We can find a complete list of system broadcast actions in the broadcast_actions.txt file in the Android SDK for each API version installed on the computer. Here you can see several API versions in the sdk folder and the platforms subfolder.



```
(base) dhcp-wifi-8021x-155-41-50-233:sdk danazh$ ls -l platforms/
total 0
drwxr-xr-x  14 danazh  staff  448 Nov 10  2020 android-27
drwxr-xr-x  14 danazh  staff  448 Aug  7  2020 android-28
drwxr-xr-x  14 danazh  staff  448 Sep  9  2020 android-29
drwxr-xr-x  15 danazh  staff  480 Sep  9  2020 android-30
drwxr-xr-x  15 danazh  staff  480 Oct 26 11:49 android-31
```

Here are some exemplary broadcast actions defined in the file.

```
android.accounts.LOGIN_ACCOUNTS_CHANGED
android.accounts.action.ACCOUNT_REMOVED
android.app.action.ACTION_PASSWORD_CHANGED
android.app.action.ACTION_PASSWORD_EXPIRING
android.app.action.ACTION_PASSWORD_FAILED
android.app.action.ACTION_PASSWORD_SUCCEEDED
android.app.action.ACTION_SHOW_NEW_USER_DISCLAIMER
android.app.action.AFFILIATED_PROFILE_TRANSFER_OWNERSHIP_COMPLETE
android.app.action.APPLICATION_DELEGATION_SCOPES_CHANGED
android.app.action.APP_BLOCK_STATE_CHANGED
android.app.action.AUTOMATIC_ZEN_RULE_STATUS_CHANGED
android.app.action.BUGREPORT_FAILED
android.app.action.BUGREPORT_SHARE
android.app.action.BUGREPORT_SHARING_DECLINED
android.app.action.CLOSE_NOTIFICATION_HANDLER_PANEL
android.app.action.COMPLIANCE_ACKNOWLEDGEMENT_REQUIRED
android.app.action.DATA_SHARING_RESTRICTION_APPLIED
android.app.action.DEVICE_ADMIN_DISABLED
android.app.action.DEVICE_ADMIN_DISABLE_REQUESTED
android.app.action.DEVICE_ADMIN_ENABLED
android.app.action.DEVICE_OWNER_CHANGED
android.app.action.DEVICE_POLICY_CONSTANTS_CHANGED
```

In the same folder, there are also activity_actions.txt and service_actions.txt defining activity action strings and service action strings respectively. These strings are also defined as Constants in the Intent class. Instead of using the literal string directly (e.g. "android.intent.action.BOOT_COMPLETED"), we can use Intent.ACTION_BOOT_COMPLETED.

Test Yourself

Where can you find all system broadcast action strings defined for API 29?

<sdk folder>/platforms/android-29/data/broadcast_actions.txt

Test Yourself

When a SMS message is received, a system broadcast message is sent. What is the action string specified in this broadcast message?

android.provider.Telephony.SMS_RECEIVED

Define a Broadcast Receiver by Subclassing BroadcastReceiver

BroadcastReceiver is an abstract Java class defined in Android to represent a broadcast receiver. To receive and handle broadcast messages, we need to first subclass BroadcastReceiver and implement the abstract method onReceive(Context!,Intent!). This method is called when some broadcast intent is received. The context and the received intent are passed into this method. We can define how to respond to the received intent here. For example, if we want to start the ProjectPortal app after the system is booted, we can register to receive the BOOT_COMPLETED broadcast intent, and start the app when this intent is received. The receiver can also handle different broadcast messages. For example, we want to show a message when the battery is charging. We can also handle the BATTARY_CHANGED intent in this receiver. The code is shown below.

```
class MyBroadcastReciever: BroadcastReceiver() {
   override fun  onReceive (context: Context, intent: Intent) {
       when  (intent.action ) {
           Intent.ACTION_BOOT_COMPLETED
           -> context.startActivity(Intent(context, LoginActivity::class .java ))
           Intent .ACTION_BATTERY_CHANGED
           -> Toast.makeText(
               context, "battery is changed" ,
               Toast.LENGTH_LONG ,
           ).show()
       }
   }
}
```

Test Yourself

To define a broadcast receiver, we need subclass ___________. It is an _____ class. It includes an abstract method ______ that must be implemented in the subclass.

BroadcastReceiver, abstract, onReceive(Context!, Intent!)

Test Yourself

To get the action string from an intent, we can use _______________.

intent.action

Registering the BroadcastReceiver

To receive the BOOT_COMPLETED and BATTERY_CHANGED intents, we also need to register this receiver. There are two ways to register the receiver: declaring it in the manifest file, or register it with a context by calling the `registerReceiver(BroadcastReceiver?, IntentFilter!)` method.

To declare it in the manifest file, use the <receiver> tag and specify its intent-filter.

```
<receiver android:name=".MyBroadcastReciever"
   android:exported= "true">
```

```
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.BATTERY_CHANGED" />
   </intent-filter>
</receiver>
```

To register it with a context, we need to first create an intent filter and then register the receiver using the registerReceiver() method call. You can also unregister it to stop receiving broadcasts by calling Context.unregisterReceiver(BroadcastReceiver!).

```
registerReceiver(MyBroadcastReciever(),
    IntentFilter().apply{
        addAction(Intent.ACTION_BATTERY_CHANGED)
        addAction(Intent.ACTION_BOOT_COMPLETED)
    })
```

Unlike manifest-registered receivers who can receive broadcast during the whole lifetime of the application, context-registered receivers (registered using the `registerReceiver`() method) only can receive broadcasts if the registering context is valid. For example, if you register within an Activity context, it can receive broadcasts only when the activity is not destroyed. If you want to receive broadcasts independent of a component life cycle, then register with the Application context. So Instead of using `this.registerReceiver(br, filter)`; We can use `getApplication().registerReceiver(br, filter)` Then it can receive broadcasts as long as the app is running.

Beginning with Android 8.0 (API level 26), the system imposes additional restrictions on manifest-declared receivers. If your app targets API level 26 or higher, you cannot use the manifest to declare a receiver for most implicit broadcasts (broadcasts that do not target your app specifically).

Intent Filters

No matter which method is used to register a broadcast receiver, we need to specify an intent filter for it. An intent filter specifies the type of intents that the associated component would like to receive. It is usually defined in the manifest using the <intent-filter> element. Inside the <intent-filter>, you can specify the type of intents to accept using one or more of these three elements, <action>, <category> or <data>. For example, a main activity (the entry activity of an app) is always associated with the following intent filter:

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

So that it can be started when an application is launched. In the above broadcast receiver example, the intent filter of the receiver specifies it can receive BOOT_COMPLETED and BATTERY_CHANGE intents. We can also create an IntentFilter object in the code as shown in the previous example.

If a component is associated with an intent filter, it can receive a certain type of intents from other applications. If no intent filter is associated with it, it can only receive an intent that explicitly sets its receiver to the component. By using intent filters, a component exposes itself to other applications.

An intent that specifies the receiving component is called an explicit intent. When using an explicit intent, the system knows exactly which component the intent should be delivered to. An intent that does not specify the receiving component, but the action (and/or data, category) is called an implicit intent. When using an implicit intent, the Android system locates an appropriate component that can respond to the intent based on its intent-filter, launches a new instance of the component if one is needed, and passes it the Intent object. Both intent filters and implicit intents should be used carefully, otherwise, potential security vulnerabilities can be exploited. We will discuss more on Android security in the next module.

Test Yourself

To register a broadcast receiver in a manifest file, we need to use ______ tag.

<receiver>

Test Yourself

To register a broadcast receiver in the code, we need to call the ______ method. To stop receiving broadcasts, we can call the _____method.

registerReceiver(), unRegisterReceiver()

Test Yourself

Which of the following methods should be used to register a broadcast receiver so that it can receive broadcasts as long as the app is running? (Check all that are true.)

Register in the manifest file.

In an activity call `this.registerReceiver(br, filter);`

In an activity call `getApplication().registerReceiver(br, filter);`

(Note: This question will mark each answer as an individual question. If a box is supposed to remain unchecked and it is left unchecked, it will mark it as correct.)

Test Yourself

Register a broadcast receiver SMSReciver to receive SMS_RECEIVED events.

```
registerReceiver(SMSReciever(),
   IntentFilter().apply{
       addAction(Intent.ACTION_SMS_RECEIVED)
     })
)
```

Sending Broadcast Messages

Besides the system can automatically send system broadcast intents, an application can also send broadcast messages by calling sendBroadcast(Intent) from a context. To restrict who can receive this broadcast intent, we can also specify permissions by using sendBroadcast(Intent intent, String Permission). Only those who have the specified permission can receive this broadcast message.

Permissions

If anyone can register to receive broadcast messages, the malicious application can also receive these messages. This can be potential security problems. Permissions are used to limit who can receive the broadcast messages. Some system broadcast messages require permissions to be received. For example, to receive a boot completed message, we need to declare RECEIVE_BOOT_COMPLETED in the manifest file. An application can also use sendBroadcast(intent: Intent!, Permission:String?) to restrict who can receive its broadcast message. For those broadcast receivers, they need to request the corresponding permissions in the manifest file using the <use-permission> tag. The manifest file for the above example is shown as follows.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="edu.bu.projectportal">

   <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
```

Test Yourself

Write a code snippet to send a broadcast message with a custom action string “edu.bu.example.action” and require the receiver to have a custom permission “edu.bu.example.MYPERMISSION”.

sendBroadcast(Intent (**"edu.bu.example.action"**), **"edu.bu.example.MYPERMISSION"**);

Topic 2: Asynchronous Processing

Processes

Each Android application runs as a separate process. A process is a program in execution. When an application runs on a device, it needs resources such as CPU, memory, file, I/O, etc. **A process is a resource consumption unit.** The operating system allocates resources to the process it runs in. **A process is also a protection unit.** To prevent applications from harming each other, each process has its own virtual address space, so one cannot access another's memory space directly. This isolation provides basic application security in Android. In the previous “Data Storage” module, we learned that each application has its own internal storage space that cannot be accessed by others. Data in memory is also protected from each other through the process model. The word “sandbox” is used to describe this isolation. Each Android application runs in its own sandbox, with its own memory address space and storage space for data.

By default, an Android application is a single process application. However, it is also possible to create a multi-process application. Additional processes can be created and defined to run different components in an application. For example, this can be done by defining the `android:process` attribute in the component to specify the process it should run in the manifest file. For example, we can define the following service component running in an additional process .myprocess.

```
<service
   android:name=" .MyRemoteService"
    android:enabled="true"
    android:process=".myprocess"/>
```

Since different processes can run simultaneously, we can improve the performance through parallelism by using multiple processes in an application. However, communication between processes is more expensive and less trivial. In most cases, different components in the same application need frequent communication and data sharing, which increases the runtime overhead and complexity when using multiple processes. The thread model is defined to address this issue.

Test Yourself

What does "each android application runs in its own sandbox" mean?

Each application has its own memory space and internal storage space that other applications cannot access. The memory isolation is achieved through the process model since each application runs in a separate process. The file storage isolation is achieved by having a private folder for each application at /data/data<appfullpackagename>

Test Yourself

What are pros and cons of a multi-process application?

Pros: isolation, parallelism. Cons: more expensive inter process communication

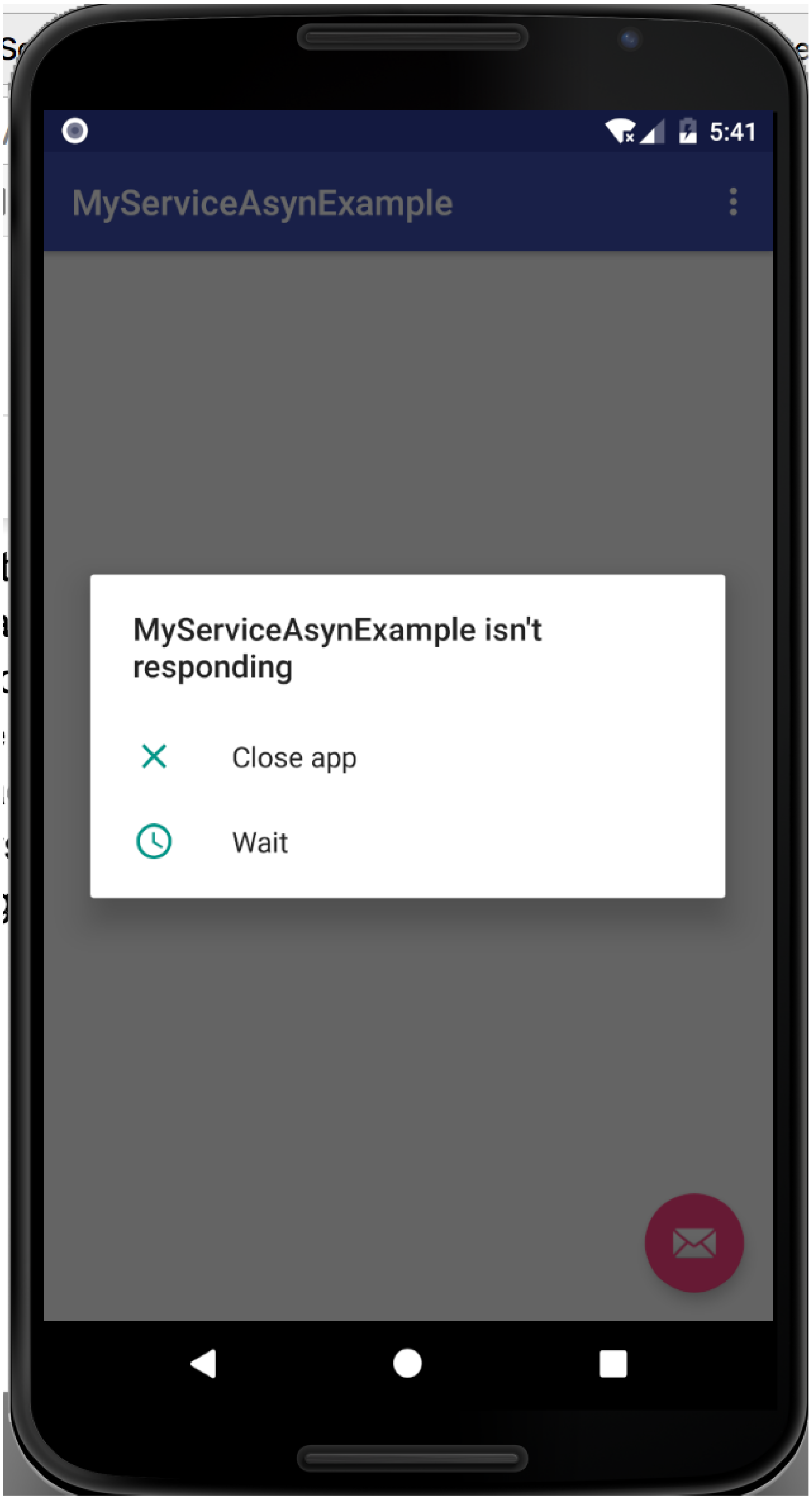
Threads

A Thread is an execution unit. Each thread runs as a separate path of execution that is managed by its own stack, the call stack which contains the information of all unreturned methods, such as parameters, local variables, return addresses, etc. Multiple threads can run at the same time to have the job done faster through parallelism. However, different from the process model, **threads in the same process share the virtual address space, thus providing no isolation.**

A process by default is a single threaded process. When we need to perform simultaneous tasks in an application, we can use a multi-threaded process. While each thread can execute independently and simultaneously, all threads within the same process share the same virtual address space. Therefore, communication and data sharing is much easier and faster. **This means that all threads can directly access code, global variables, constants, and objects in the heap. But they have their own stack (e.g. local variables, method parameters), and can be scheduled independently.** Because multiple threads can access the same data, correctly handling synchronization is crucial in developing a multi-threaded application.

Android Application Threads

An Android application runs by default as a single threaded process. That means all components are running in the same thread of the same process. No parallelism is provided by default. All run sequentially. This thread is called the **main thread**. When an activity is created, it runs in the main thread of the application. If we need more parallelism, we need to explicitly create new threads. However, the newly created threads cannot run UI elements such as activities. All the commands issued by the Android operating system, such as onClick, onCreate, etc., are sent to and processed by this main thread. Therefore, **a main thread is also called a UI thread**. To provide better user experience, the UI thread should not be blocked by any other long run tasks, so that the user interaction can be responded to immediately. Any time-intensive operations such as networking should not block the main thread. These operations should be done asynchronously in separate threads. When a long operation (e.g. longer than a few seconds) is performed on the UI thread, the application may be too busy to respond to messages sent by the Android operating system. The user may be presented with the infamous "application not responding" (ANR) dialog. The user might then decide to quit your application and uninstall it if they are unhappy. For example, the following screenshot shows an ANR dialog.



Some common blocking operations include:

- Accessing networking resources
- Accessing large data set
- Accessing large Database
- Accessing large files such as multimedia files
- Lengthy and complex calculations

These operations should be executed in a different thread so that they will not block the UI thread. How to create new threads to run these operations? In this module, we will discuss a number of different ways to perform the asynchronous processing.

Java Thread API

The first way is to create new threads using native Java thread APIs. Java provides two ways to implement threads. One is directly using the Runnable interface, the other is to subclass the Thread class.

The Runnable interface is an interface in Java defining a single abstract method run(). It abstracts a unit of executable code. We can create a thread and pass a runnable object as the parameter. The following code creates a runnable object using an anonymous class expression.

```java
public class MyActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       try {
           Thread t1 = new Thread(new Runnable() {
               public void run() {
                   //code to perform in the background thread
               }
           });
           t1.start();
       }catch(Exception e) {
           Log.d("Thread", "error in threads");
       }
   }
}
```

We can also subclass the Thread class and implement the `run()` method.

```java
public class MyActivity extends Activity {
   @Override
   public void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);
       try {
           Thread t1 = new MyThread();
```

```
        }catch(Exception e) {
            Log.d("Thread", "error in threads");
        }
    }
}

public class MyThread extends Thread {
    @Override
    public void run() {
        //code to perform in the background thread
    }
}
```

While both methods can be used to create a thread, the first one is more prefered. This is aligned with one of the design principles: “favor composition over inheritance when reusing the implementation.”

Test Yourself

All components in an Android application run in a _____ thread of ______ process by default.

single, single

Test Yourself

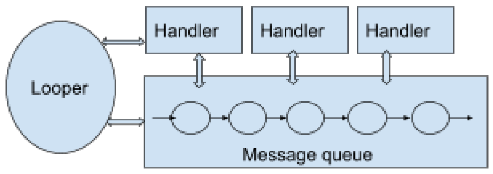
The main thread is also called the ____ thread. All UI elements run in the _____ thread. Additional threads ____ run UI elements.

UI, UI, cannot

Handlers and Handler Threads

Handlers and Handler Threads When we use raw Java thread APIs to create a new thread, we need to explicitly implement synchronization in application threads when they access (write) the same data. This can be very error prone. Therefore, we usually don’t use Java threads directly in the Android app. Instead, Android provides several asynchronous processing APIs that will do the synchronization for you transparently. Handlers can be used to communicate between threads.

The idea is that each thread can be associated with a message queue, and a message looper to get messages from the queue. The message queue is synchronized by the Android system, so no explicit synchronization code is needed at the application level. Instead, we can simply put or get messages from the message queue in our application code. Handlers and a looper are defined to access the message queue and handle the messages associated with a thread.



Looper is the class defined in Android to manage a message queue for a thread. Each looper is associated with a thread and a message queue. The main thread is already associated with a looper. We can get the main thread looper by calling `Loop.getMainLooper()`to get the main looper for the main thread. A new thread created by native Java Thread APIs do not have a message loop associated with them. To create a looper for it, we can call `Loop.prepare()` in the thread that is to run the loop, and then use `Loop.loop()` to have it process messages until the loop is stopped. HandlerThread in Android is a subclass of the Java Thread class. It is a handy class for starting a new thread that has a looper. If we need to create a thread to communicate with other threads, **it is better to create a HandlerThread object instead of a native Java Thread object, so that we can send and receive messages to/from other threads**.

Putting a message into the message queue or handling a received message in a thread is not directly through a looper, instead through a handler. Handler is a class defined in the android.os package to send or process messages. Each Handler instance is associated with a looper, thus associated with a single thread and that thread's message queue. However, We can have many handlers associated with a message looper in a thread. To use a handler, we need to explicitly create a handler object and specify what messages to put into the message queue, or how to handle a received message. **A Handler object** created on the UI thread exposes a **thread-safe** message queue on which background threads can asynchronously add either messages or requests for foreground runnables to act on their behalf.

Beside normal messages (defined by Message objects which contain data defined by Bundle objects), a handler can also put Runnable objects in the message queue. When the looper gets a runnable from the message queue, it will execute the code specified by the runnable.

To create a handler, we can call one of the Handler constructor methods. By default, the handler is created for the current thread where the method is called, and associated with the default looper of the thread. We can also specify the looper in the constructor method to associate with. To define how to handle received messages, we need either create a subclass of Handler, and override the handleMessage() method, or pass the callback method through the constructor method.

Public constructors
Handler() Default constructor associates this handler with the **Looper** for the current thread.
Handler(callback: **Handler.Callback**?) Constructor associates this handler with the **Looper** for the current thread and takes a callback interface in which you can handle messages.
Handler(looper: **Looper**) Use the provided **Looper** instead of the default one.
Handler(looper: **Looper**, callback: **Handler.Callback**?) Use the provided **Looper** instead of the default one and take a callback interface in which to handle messages.

For example, the following code shows how to create a handler for a handle thread using the constructor method. The first parameter is the looper of the handler thread. Without this parameter, the looper is the default one, which is the main looper of the main thread. The second parameter is the callback to define a single method handleMessage() to handle the received message. Therefore, a handler is created using the ***receiver thread***'s looper, but it can be used by the sender to send a message to this handler.

```
handleMessage(msg: Message): abstract Boolean
```

```
myHandler = Handler(myHandlerThread!!.looper, Handler.Callback() {msg->
   // handle the received message (msg)
       true
});
```

Then another thread can use this handler to send messages to the main thread. The message received at the main thread will be handled using the defined `handleMessage`() method.

Several methods are defined to send a message. We can send a message immediately or at a future time.

- sendMessage(Message),
- sendMessageAtTime(Message, Long)
- sendMessageDelayed(Message, Long)

We can also use a handler to post a runnable immediately or at a future time.

- post(Runnable),
- postAtTime(Runnable, Long)
- postDelayed(Runnable,long)

The following code snippet shows an example of creating a handler thread to do the long operation and communicating with the main thread through a handler.

```
class MainActivity : AppCompatActivity() {
   private lateinit var mainHandler: Handler
   private lateinit var threadHandler : Handler
   private lateinit var mHandlerThread: HandlerThread

   private lateinit var binding:ActivityMainBinding

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       binding = ActivityMainBinding.inflate(layoutInflater)
       setContentView(binding.root)

       binding.displayText.setText("")

       // create a handler for the main thread
       mainHandler = Handler()

       // create a handler thread mHandlerThread
       mHandlerThread = HandlerThread("background handler thread")
       // start mHandlerThread  by calling its start() method
       mHandlerThread!!.start()
```

```
        // Create a threadHandler that is associated with the mHandlerThread,
        threadHandler = Handler(mHandlerThread!!.looper, Handler.Callback{msg->
           // run the long operation, and send back the result to the main thread
            val i = LongOperation.run(msg.getData().getLong("delay"))
            mainHandler!!.post{
                binding.displayText.text = "Thread " +
                        Thread.currentThread().id +
                        " Done : i =" + i
            }
            // After 2 seconds, the output is cleared
            mainHandler!!.postDelayed({ binding.displayText!!.text = "" },
                2000)
            true
        })
    }

    // called when clicking the display button
    fun display(v: View?) {
        // get the delay value from the editview
        val delay = binding.delayTime!!.text.toString().toLong()

        binding.displayText!!.text = """execute
            for (i = 0; i <${delay * 100000000}; i++) """

        val bundle = Bundle().apply {
            putLong("delay", delay)
        }
        // send the delay to the handler thread
        threadHandler!!.sendMessage(Message.obtain().apply{
            setData(bundle)})
    }
}
```

Here, we create a separate handler thread to execute a long operation. We also create a handler for the main thread using the default looper, and a handler for the new handler thread. When the user inputs a delay time, the main thread will send this delay time to the handler thread using `threadHandler!!.sendMessage()`. Once the handler thread receives it, it extracts the delay time from the message and executes a long operation. Once it is finished, it asks the main thread to display the result by posting a runnable to the main thread.

Test Yourself

If a thread t1 wants to send a message to the main thread, a handler myHandler should be created for _____ . ________ should call myHandler.sendMessage(). If the main thread needs to send a message to t1, then a handler should be created for ____. ________ should call myHandler.sendMessage().

A. The main thread

B. t1

A, B, B, A

Test Yourself

How would you make an operation be executed at a later time?

Create a handler and call its postDelayed(Runnable, Long). The operation should be encapsulated in the Runnable object.

Kotlin Coroutine

Kotlin provides a flexible, safer and less error-prone way to handle asynchronous processing at the language level: coroutines. kotlinx.coroutines is a rich library for coroutines developed by JetBrains. Coroutines is the recommended solution for asynchronous programming on Android. It is lightweight with fewer memory leaks. It has built-in cancellation support and is supported by many Jetpack libraries.

Suspend

One of the core concepts in Kotlin coroutines is suspendable computation. A suspending function is a function that could be started, paused, and resumed. A coroutine is an instance of suspendable computation. It is conceptually similar to a thread, in the sense that it takes a block of code to run that works concurrently with the rest of the code. However, a coroutine is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

To define a suspending function, simply add the keyword “suspend” in front of the function definition. A suspending function can only be called by another suspending function or launched using a coroutine builder such as launch.

```
interface ProjectDao {
   @Insert(onConflict = OnConflictStrategy.REPLACE)
   suspend fun addProject(project:Project)
   @Delete
   suspend fun delProject(project:Project)

   . . .
}
```

```
class ProjectRepository(
  private val projectDao: ProjectDao){
   . . .
```

```
    suspend fun addProject(project: Project){
                projectDao.addProject(project)
    }
. . .
}
```

Launch

Launch is a function that creates a coroutine and dispatches the execution of its function body to the corresponding dispatcher. It launches a new coroutine without blocking the current thread and returns a reference to the coroutine as a Job. The coroutine is canceled when the resulting job is canceled.

Coroutines follow a principle of structured concurrency which means that new coroutines can be only launched in a specific CoroutineScope which delimits the lifetime of the coroutine. In a real application, you will be launching a lot of coroutines. Structured concurrency ensures that they are not lost and do not leak. An outer scope cannot complete until all its children coroutines complete. Structured concurrency also ensures that any errors in the code are properly reported and are never lost.

It is common to start a coroutine inside a ViewModel class with **ViewModelScope** defined by Android. This enables the coroutine to be executed in the scope of the ViewModel. If the ViewModel is destroyed for any reason (e.g the user , viewModelScope is automatically canceled and all running coroutines are canceled as well.

When using the launch function, ***a coroutine dispatcher*** is provided to determine what thread or threads the corresponding coroutine uses for its execution. In Kotlin, all coroutines must run in a dispatcher, even when they're running on the main thread. Coroutines can suspend themselves, and the dispatcher is responsible for resuming them. To specify where the coroutines should run, Kotlin provides three dispatchers that you can use:

- Dispatchers.Main - Use this dispatcher to run a coroutine on the main Android thread. This should be used only for interacting with the UI and performing quick work. Examples include calling suspend functions, running Android UI framework operations, and updating LiveData objects.
- Dispatchers.IO - This dispatcher is optimized to perform disk or network I/O outside of the main thread. Examples include using the Room component, reading from or writing to files, and running any network operations.
- Dispatchers.Default - This dispatcher is optimized to perform CPU-intensive work outside of the main thread. Example use cases include sorting a list and parsing JSON.

```
fun addProject (project: Project){
    viewModelScope.launch (Dispatchers.IO ) {
        projectRepository.addProject(project)
    }
}
```

Async & Wait

When using launch, a job is returned, but no resulting value is returned. If we want to get some data directly from the suspending function, we can use async. It is similar to launch, in terms that it also starts a separate coroutine that works concurrently with other code, but different from launch, in terms that it returns a Deferred, a job that can carry the returning result. Then we can use .await() on a deferred value to get its result. Suppose in the ProjectRepository class, we have, where loadAllProjects() returns a list of projects.

```
suspend fun loadAllProjects(): List<Project> {
   return projectDao.loadAllProjects()
}
```

We can use async() to get a deferrable job and use await() to get the returned result running in a suspending function. Be aware that await() is also a suspending function that needs to be launched.

```
fun reloadAllProjects() {
   val bgJob = viewModelScope.async {
           projectRepository.getAllProjects()
       }
   viewModelScope.launch {
       allProjectsLiveData.value = bgJob.await()
   }
}
```

Write Asynchronous ROOM DAO

As database operations may involve long io operations, Room doesn't allow the Dao methods executing directly on the main thread to prevent them from blocking the UI. All Dao methods should be asynchronous. You can find several different options to handle Room Dao methods at Write asynchronous DAO queries . In our Project Portal example, we mainly use Kotlin Coroutines and LiveData.

- For one-time write operations, use Kotlin suspend functions.
- For one-time read operations, use Kotlin suspend functions.
- For Observable read operations, use JetPack LiveData.. Room generates all the necessary code to update the LiveData object whenever a database is updated. The generated code runs the query asynchronously on a background thread when needed.

Test Yourself

How would you define a suspending function?

Add the “suspend” keyword in front of the function definition.

Test Yourself

What does the launch function do?

start a coroutine

Test Yourself

If using Dispatchers.Main, which thread will the coroutine run on?

the main thread

Test Yourself

Why use the async function?

get the returned value from an asynchronous function

Topic 3: Service

Services

Services are application components without user interfaces. They are usually used to perform long-running operations in the background. It can exist even if the user switches to another application. It has a totally different life cycle than an activity. **Be aware that a service is not another background process or background thread.** It is just an Android component. Since an Android application runs in a single threaded process by default, all services are also running in the main thread as activities. However, services usually perform long run operations such as heavy calculation, long initialization, networking, database operations, which can potentially block the UI thread. **Therefore it is common that services are executed in different threads from the main thread.** We can use some asynchronous processing APIs discussed in the previous section to execute long run operations in the service. Android also provides several other service related asynchronous processing APIs.

There are several types of services in Android:

- **A background service** performs some operation that isn't directly noticed by the user.
- **A foreground service** performs some operation that is noticeable to the user. Foreground services must display notifications. Foreground services continue running even when the user isn't interacting with the app.
- **A bound service** offers a client-server interface that allows components to interact with the service, send requests, receive results, and even do so across processes with interprocess communication (IPC).

The Service Class and Intent

Service is an abstract class defined in Android to support background processing. It is an indirect subclass of android.content.Context, so it inherits all public methods defined in Context, such as sending an intent. It **includes an abstract method onBind() that must be implemented** in its subclass. To create a service class, we need to subclass Service, and implement `onBind()` and mostly override several other callbacks.

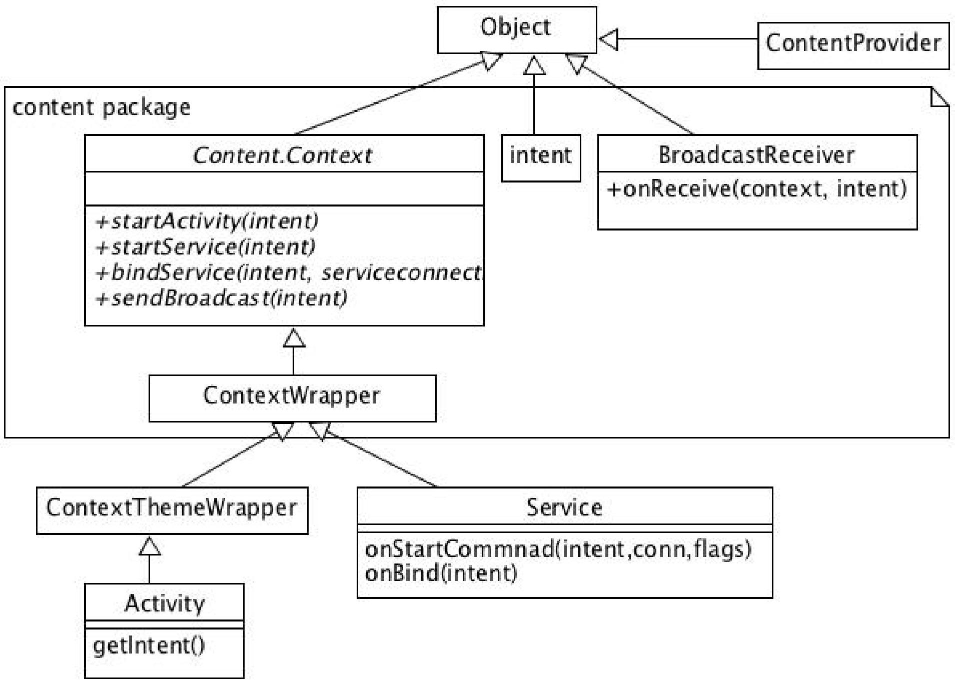
- `onCreate()`: called when the service is initially created. Override this method to do some extra initialization work.
- `onDestroy()`: called when the service is destroyed. Override this method to clean up any resources such as threads.
- `onStartCommand(Intent!, Int, Int):Int`: called when the service is started by another component using startService(). Override this method to process the intent received through startService(). When the work is done, call stopService()/stopSelf().
- `onBind(Intent!):IBinder?`: called when another component wants to bind with the service through bindService(). The communication channel is returned. This method must be implemented in the subclass. If no communication is needed, we can simply return null in this method.

To start a service we can call `startService()` in another component such as an activity. It in turn will call `onStartCommand()` of the target service. If the service is not created, it will first create a service instance and call `onCreate()`. A service started by `startService()` is also called started service.

A service can be started by an activity, or another service. This is done by sending an intent to the target service. This `startService()` method is used to send an intent to a started service object. We can also use the `bindService()` method to send an intent to a bounded service (and create it if needed). So far, we have learned several methods to send an intent. These methods are all defined in the Context class. So any context object can call the following methods.

- Send an intent to an activity: `startActivity(intent!)`
- Send an intent to a service: `startService(intent!)`, `bindService(intent!)`
- Send a broadcast intent to broadcast receivers: `sendBroadcast(intent!)`

Besides the receiving component name, we can also include the action string and additional data in an intent. To process an intent, an activity object uses the `intent` member field or calls `getIntent()` to get the received intent, a service object calls `onStartCommand(Intent!, Int, Int)` or `onBind(intent)` where the received intent is passed in, and the broadcast receiver calls `onReceive(context, intent)` where the received intent is passed in. The following class diagram shows all Android component classes and methods related to intents. **To ensure that your app is secure, always use an explicit intent when starting a Service** and **don't declare intent filters for your services**.



As other Android components, we also need to **declare all service components in the manifest** file inside the <application> element using the <service> tag. For example,

```
<service
    android:name=".Service.MyBgServiceST"/>
```

Test Yourself

What is the difference between a service and an activity?

An activity is an entry point for a UI screen, and a service is a component without UI.

Test Yourself

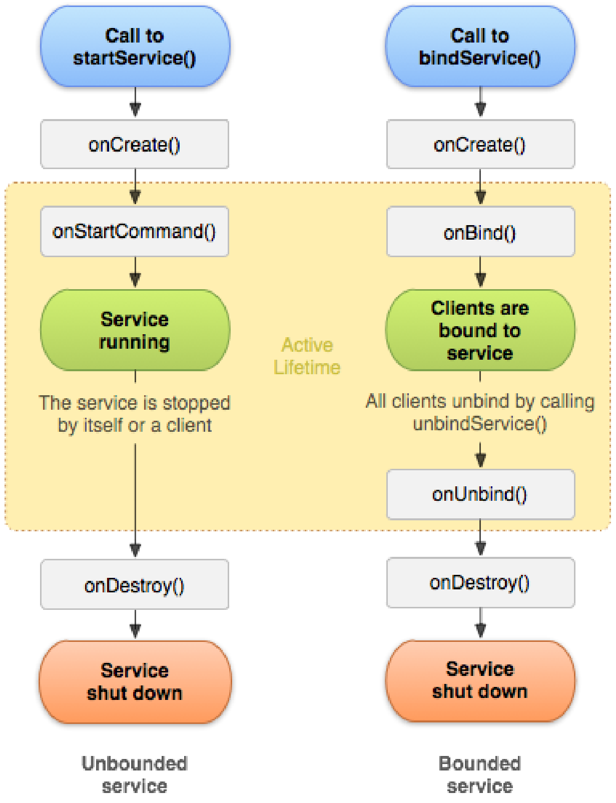
Which method in the Service class is an abstract method?

onBind()

Service Lifecycle

The following diagrams from the Android developer website show the lifecycle of a service. The left side diagram shows the life cycle of a started service (created by clients calling `startService()`). The right side diagram shows the life cycle of a bound service (created by clients calling `bindService()`). Though we have discussed the started services and bound service separately, a service can be both. As long as `onBind()` is implemented and returns a valid IBinder object (not null), clients can always call `bindService()` to bind to it.

If multiple components call `startService()` on the same service, the first one will create and start the service, and the system will only call onCreate() once. Then `onStartCommand()` is called multiple times to handle each intent. The service can be stopped by itself (usually calling `stopSelf()` in `onStartCommand()`) or stopped by the client. If multiple components call `bindService()` on the same service, the first one will create and start the service, and the system will call `onCreate()` once. The `onBind()` can be called by multiple clients, returning different ICC channels to each client. The client can then `unBindservice()`. When all clients unbind to the service, the system will destroy the service.



Services overview

Test Yourself

Which methods are only called once in the service's life cycle? Which methods can be called multiple times?

once: onCreate(), onDestroy(), multiple times: onStartCommand(), onBind(), onUnBind()

Test Yourself

When is onStartCommand() called? When is onBind() is called?

When a client calls startService(), onStartCommand() is called. When a client calls bindService(), onBind() is called.

Background Services

A background service is a service performing some operation that isn't directly noticed by the user. For example, the following code snippet defines a background service running in the main thread. Be aware that since Service is an indirect subclass of Context, we can pass `this` to the Toast.makeText() method. Also, since a service runs in the main (UI) thread by default, we can still use Toast here to display a toast message even if the service is a background service. However, we cannot display a Toast message in a different thread because all UI elements are only handled in the UI thread. For a background service, we need to override `onStratCommand()`,where the background work should be included. If your app targets API level 26 or higher, the system imposes restrictions on running background services when the app itself isn't in the foreground.

```
class MyBgServiceST  //empty constructor
  : Service() {
   override fun onBind(intent: Intent): IBinder? {
       return null
   }

   override fun onStartCommand(intent: Intent, flags: Int, startId: Int): Int {
       Log.d(TAG, "Service onStartCommand ")

       //some background operation here

       Toast.makeText(this, "service started", Toast.LENGTH_LONG).show()
       stopSelf()
       return START_NOT_STICKY
   }

   companion object {
       private const val TAG = "BgServiceSameThread"
   }
}
```

To start this service we can call `startService()` in another component such as an activity. `startService(Intent(this, MyBgServiceST::class.java))`

Test Yourself

Which method needs to be overridden for a background service?

onStartCommand()

Test Yourself

How to start a background service?

call startService() from any context object

Foreground Service

A foreground service is noticeable to the users. Foreground services must display notifications, so that the user is aware of them. For example, an audio app would use a foreground service to play an audio track, so the user is aware of it and the user usually can play, pause or stop the audio through the notification instead of opening the app itself. Similar to background services, foreground services continue running even when the user isn't interacting with the app.

Since the user is not aware of background services, malicious code can be executed in the background service without the user's notice. To improve the security, Android O imposes some restrictions on running background services when the app itself isn't in the foreground. In general, background services should be used with extra caution. Alternatively, we can use foreground services or scheduled jobs.

To define a foreground service, we need to implement startForeground(int, android.app.Notification) in the service subclass. A started service (started by `startService(Intent)`) can call this method to put the service in the foreground, supplying the ongoing notification to be shown to the user while in this state. We will introduce notifications and provide examples of foreground services in the next module.

Test Yourself

What makes a background service a foreground service??

Adding a notification.

Bound Services

A bound service allows other application components (such as an activity) or components from a different application to bind to it so that they can communicate with each other.

To define a bound service, we need to implement the `onBind()` callback method to return an IBinder that defines the interface for communication with the service. IBinder is the base interface describing the abstract protocol for interacting with a remotable object. A common way is to create a Binder object and return it to the caller.

Local Bound Services

Here shows how to define a local bound service. Mainly we need to implement the `onBind()` method. Since this service is only used by local clients, which are application components from the same application, we can simply return the service object itself through `onBind()`, instead of creating an IPC (Interprocess Communication) Channel. To return an IBinder object from `onBind()`, we define a `Binder` subclass, `MyLocalBinder`, and return the service object itself.

After obtaining the service object, the client can call any public methods defined in the service. In this example, the client can get the `currentTime` defined in the service.

```
class MyLocalBoundService : Service() {
   private val myBinder : IBinder = MyLocalBinder()
    private var bound = false

   override fun onBind(intent: Intent): IBinder? {
       Log.d(TAG, "service onBind")
       bound = true
       return myBinder
   }

   override fun onUnbind(intent: Intent): Boolean {
       Log.d(TAG, "service onUnBind")
       bound = false
       return true
   }

   inner class MyLocalBinder : Binder() {
       val service: MyLocalBoundService
           get() = this@MyLocalBoundService
   }

   @get:RequiresApi(api = Build.VERSION_CODES.N)
   val currentTime:String
        get() {
            if (bound) {
               val dateFormat = SimpleDateFormat("HH:mm:ss MM/dd/yyyy", Locale.US)
                return dateFormat.format(Date())
           }
            return ""
       }

    companion object {
        private const val TAG = "LocalBoundService"
   }
}
```

An application component (either from the same application or different application) binds to a service by calling `bindService()` to retrieve the IBinder object returned by `onBind()`, so that they can communicate with each other using the IBinder object. The service is created if needed.

When calling `bindService(intent, conn, flags)`, besides the intent that specifies the target service, we also need to pass a ServiceConnection object. ServiceConnection is an interface defined with two abstract callback methods: `onServiceConnected(ComponentName,IBinder)`, and `onServiceDisconnected(ComponentName)`.

We need to create a ServiceConnect object and implement these two callback methods. The `onServiceConnected()` method is called when a connection to the service has been established, and the `IBinder` of the communication channel to the service is passed into this method. The `onServiceDisconnected()` is called when a connection to the service has been lost. This typically happens when the process hosting the service has crashed or been killed. This does not remove the ServiceConnection object itself -- this binding to the service will remain active, and you will receive a call to `onServiceConnected()` when the service runs again.

The following code snippet shows how a client binds to a local bound service defined above. Since this is a local bound service, only components from the same application can bind to it. Because they run in the same process, thus the same address space, so that the client can access the service object directly in the heap. If the client is from another application, it cannot access the service object directly because it is in a different process, thus a different address space. In that case, we need to create an IPC channel. We will talk about remote bound service next. In this example, an activity in the same application binds to the service. To call `bindService()`, we need to create a ServiceConnection object and implement these two callback methods.

```
fun startLocalBoundService(v: View?) {

   //create a service connection to monitor the connection with the boundservice
   localConn = object : ServiceConnection {
       override fun onServiceConnected(name: ComponentName, service: IBinder) {
           val binder: MyLocalBoundService.MyLocalBinder = service as MyLocalBoundService.
           myLocalBoundService = binder.service
           isLocalBound = true
           Log.d(TAG, "local service is connected")
       }

       override fun onServiceDisconnected(name: ComponentName) {
           isLocalBound = false
           myLocalBoundService = null
           Log.d(TAG, "local service is disconnected")
       }
   }
   val intent = Intent(this, MyLocalBoundService::class.java)
   bindService(intent, localConn as ServiceConnection, Context.BIND_AUTO_CREATE)
}
```

Then the client can use this service and call any public methods defined in this service.

```
fun showTime(v: View?) {
   binding.textView.text =
       myLocalBoundService?.currentTime
}
```

Remote Bound Services

The following code shows how to define a remote bound service. Here we need to use a Messenger object to create an IPC channel. Messenger is a final class defined in Android, pointing to a handler which is used to send messages to it. The implementation underneath is just a simple wrapper around a Binder object that is used to perform the communication. A Messenger object can be constructed by a given handler, or a raw IBinder object. It provides two constructor methods:

- `Messenger(Handler!)`: Create a new Messenger pointing to the given Handler
- `Messenger(IBinder!)`: Create a Messenger from a raw IBinder,

In the following code snippet, we create a Messenger object pointing to an IncomingHandler object that handles incoming messages. Then we can get the `IBinder` object that this Messenger is using to communicate with its associated Handler. When the client gets the IBinder object, it can reconstruct the Messenger object, and call its `send(Message)` method to send messages to the service.

```
class MyRemoteBoundService : Service() {
   val myMessenger: Messenger = Messenger(IncomingHandler())



   override fun onBind(intent: Intent): IBinder? {
       Log.d(TAG, "service onBind")
       return myMessenger.binder
   }

   override fun onUnbind(intent: Intent): Boolean {
       Log.d(TAG, "service onUnBind")
       return false
   }

   internal inner class IncomingHandler : Handler() {
       override fun handleMessage(msg: Message) {
           val data = msg.data
           val dataString = data.getString("MyString")
           Log.i(TAG, "handle message")
           Toast.makeText(applicationContext, dataString, Toast.LENGTH_LONG).show()
       }
   }

   companion object {
       private const val TAG = "RemoteBoundService"
   }
}
```

In the client, we call bindService() to bind to the service. Through the ServiceConnection object, it can return the IBinder object, and reconstruct a Messenger object.

```
fun startRemoteBoundService(v: View?) {
    remoteConn = object : ServiceConnection {
        override fun onServiceConnected(name: ComponentName, serviceBinder: IBinder) {
            myMessenger = Messenger(serviceBinder)
            isRemoteBound = true
            Log.d(TAG, "remote service is connected")
        }

        override fun onServiceDisconnected(name: ComponentName) {
            isRemoteBound = false
            Log.d(TAG, "remote service is disconnected")
        }
    }
    val intent = Intent(this, MyRemoteBoundService::class.java)
    bindService(intent, remoteConn as ServiceConnection, Context.BIND_AUTO_CREATE)
}
```

Then we can use the Messenger object to send messages to the service.

```
fun sendMsg(v: View?) {
    if (!isRemoteBound) return
    val msg = Message.obtain()
    val bundle = Bundle()
    bundle.putString("MyString", "Message Received")
    msg.data = bundle
    try {
        myMessenger?.send(msg)
    } catch (e: RemoteException) {
        e.printStackTrace()
    }
}
```

After the communication is done, the client can call `unBindService()` to unbind. Multiple clients can bind to the same service simultaneously. When there are no clients bound to the service, the system destroys the service.

In addition to using Messenger, we can also use AIDL (Android Interface Definition Language) to define IPC channels. This is more complicated and we will not cover it here.

Let us summarize the steps to use a bound service:

In the client (who call `bindService()`):

- create a ServiceConnection object that implements two callbacks (`onServiceConnected(), onServiceDisconnected()`)
- In `onServiceConnected()`, the reference to the binder which will be used to communicate with the service is passed in.

- To bind a remote service (the client and service are not in the same process), create a Messenger that wraps around IBinder, which can be used to send messages later.
- Call `bindService()` and pass the ServiceConnection object into the method.
- Get the returned IBinder object and use it to communicate with the service.
- Call `unBindService()` to unbind after all is done.

In the service:

- Implement `onBind()`. Create an IBinder object and return it.
- In the remote service, create a Messenger by creating a handler that implements `handlerMessage()`.
- Implement onUnBind() if needed.

Here is the whole workflow:

1. The client creates a ServiceConnection object.
2. The client calls `bindService()`, passing the ServiceConnection object.
3. The system sends the intent to the service. If the service is not created yet, the system creates the service. The service `onCreate()` is called.
4. The service calls `onBind()`, creates an IBinder object and returns it.
5. The connection is established. The client `onServiceConnected()` is called.
6. The client can get the returned IBinder object.
7. The client uses the IBinder object to communicate with the service.
8. When all is done, the client calls `unBindService()`. The service `onUnBind()` is called.
9. After all clients unbind, the service is destroyed, and `onDestory()` is called.

Test Yourself

How would you construct an IBinder object for a local bound service?

We can extend Binder, and return the service object itself through a getter method.

Test Yourself

How would you construct an IBinder object for a remote bound service?

We can use a Messenger. First creating a Handler object and then create a Messenger object pointing to it, finally get IBinder object from the messenger by calling getBinder().

Test Yourself

Describe the order of all methods called in the service and client?

bindService() - > onCreate() -> onBind() -> onServiceConnected() -> unBindService() -> onUnBind() -> onDestroy()

Job Services and Job Schedulers

Since the service component is designed to perform some work in the background without the application UI, and it is also running in the main thread, it is important to:

1. Properly schedule these work in the background.
2. Executing any long operations in separate threads.

Android provides additional APIs to help manage services. JobService and JobScheduler are added in API level 21 to help schedule services.

Class: JobService

JobService is a subclass of Service. It is the base class that **handles asynchronous requests that were previously scheduled**. It is an abstract class with two abstract methods declared: `onStartJob(JobParameters)` and `onStopJob(JobParameters)`. To use JobService, we need to subclass it and implement these two abstract methods. In particular, we need to implement our job logic in `onStartJob(JobParameters)`. The JobParameters specify how and when the job service can be scheduled. This service executes each incoming job on a separate handler, and thus the job can be scheduled in the future.

```
class MyJobService : JobService() {
   override fun onStartJob(params: JobParameters): Boolean {
       val id = params.extras.getInt("id")
       Toast.makeText(this, "Job $id is scheduled", Toast.LENGTH_LONG).show()

        // do some work here

       return false
   }

   override fun onStopJob(params: JobParameters): Boolean {
       Toast.makeText(this, "Job is finished", Toast.LENGTH_LONG).show()
       return false
   }
}
```

However, be aware that this service itself does not provide any additional thread support. It executes each incoming job on a android.os.Handler running on your application's main thread. This means that you must execute your long operation using an asynchronous processing of your choosing.

Class: JobScheduler

JobService is the entry point for the callback from the JobScheduler, which we use to schedule all jobs. JobScheduler is an abstract class defined in Android for scheduling various types of jobs to be executed in the application's own process. To use JobScheduler, we do not instantiate it directly, instead we can obtain a JobScheduler object from the context through `Context.getSystemService(Context.JOB_SCHEDULER_SERVICE)`. Then call its schedule(JobInfo) and pass a JobInfo object to schedule the job.

JobInfo is a class defined in Android to represent a job info that can be scheduled by the JobScheduler. A JobInfo object is constructed by a JobInfo.Builder object. To create a JobIno.Builder object, we need to specify the component name of the Job Service that will be scheduled by the JobScheduler . Then we can set a number of parameters and constraints on the JobInfo object, including the scheduling period, the scheduling delay, scheduling conditions , etc. You must specify at least one sort of constraint.

```
fun scheduleJob(v: View?) {
       val jobId = 0
       val myJobInfo: JobInfo =
           JobInfo.Builder(
           jobId,
           ComponentName(this, MyJobService::class.java))
               .apply {
                   setRequiresBatteryNotLow(true)
                   setExtras(PersistableBundle().apply{putInt("id",jobId)})
               }.build()
       val scheduler: JobScheduler =
               getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler
           scheduler.schedule(myJobInfo)
   }
```

Android O starts to impose some restrictions on the background service to help improve system performance. When you need a long run background service, JobService and JobScheduler are recommended to use. The JobScheduler is guaranteed to get your job done, and it can intelligently schedule your background job to optimize the system performance.

WorkManager

There are other Android APIs introduced to ease the asynchronous processing in the service, such as IntentService andJobIntentService. However they are deprecated now in favor of the new WorkManager APIs introduced in the Android JetPack. Workmanager is the primary recommended API for all of the background processing as the work is persistent through app restarts and system reboots. It provides backward compatibility to API level 14, and replaces all the existing background scheduling APIs including FirebaseJobDispatcher, GcmNetworkManager and JobScheduler.

The following class diagram shows several related classes and their relationships. The basic persistent work is defined by a Worker class to specify the work to do. The WorkRequest is the class to request either a one time or a periodic work with the specific worker and constraints. The WorkManager class is used to schedule all work requests. The Worker class is an abstract class defining an abstract method doWork(), which runs asynchronously on a background thread provided by WorkManager. We need to create our own Worker class by subclassing the Worker class and implementing the doWork() method.

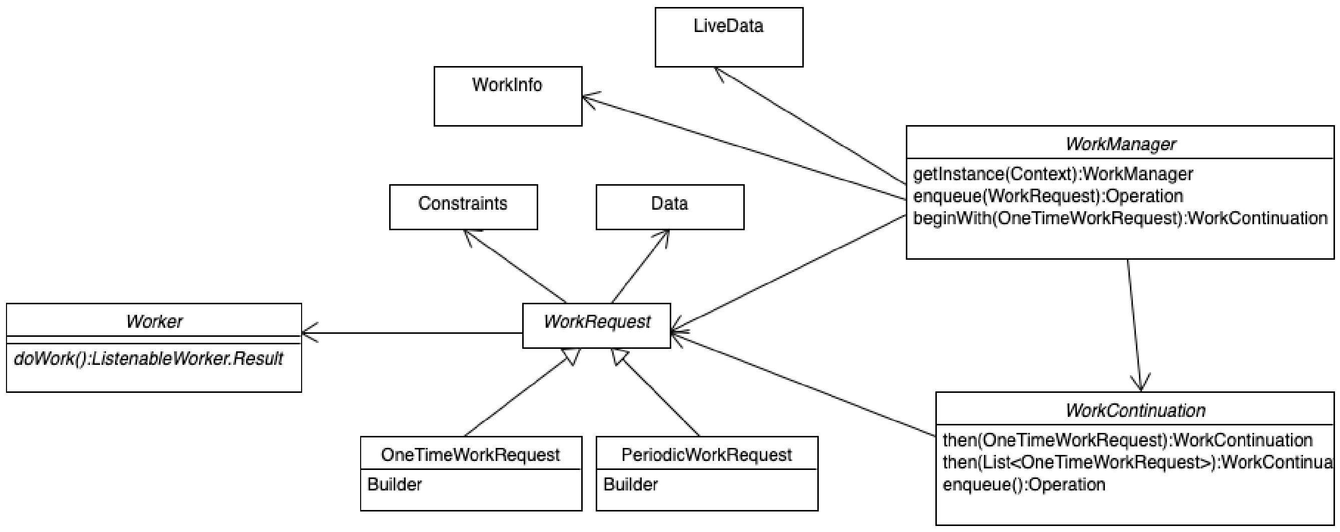
```
class MyWorker1 (context: Context,
                workerParameters: WorkerParameters
) : Worker(context, workerParameters) {
   override fun doWork(): Result {
       // your work here

       return Result.success(outputData)
   }
}
```

The WorkRquest class is also an abstract class which is the base class for specifying parameters for work that should be enqueued in WorkManager. There are two concrete implementations of this class: OneTimeWorkRequest and PeriodicWorkRequest. After creating a worker class, we need to create a work request using a builder of either a OneTimeWorkRequest or PeriodicWorkRequest. We can set the constraints and input data for the work request.

```
private val myWork1Request = OneTimeWorkRequest.Builder(MyWorker1::class.java)
    .setConstraints(networkConstraints)
    .setInputData(getInputData("id", "id1")).build()
```

Finally, we will schedule the work using a work manager. The WorkManager is also an abstract class. We need to first get an instance of the WorkManager class, and then simply enque each work request. We can also specify a chain of work requests.

```
private val workManager = WorkManager.getInstance(context)
workManager.beginWith(myWork1Request)
    .then(myWork2Request)
    .enqueue()
```


LiveData
WorkInfo
WorkManager
getInstance(Context):WorkManager
enqueue(WorkRequest):Operation
beginWith(OneTimeWorkRequest):WorkContinuation
Constraints
Data
Worker
doWork():ListenableWorker.Result
WorkRequest
OneTimeWorkRequest
Builder
PeriodicWorkRequest
Builder
WorkContinuation
then(OneTimeWorkRequest):WorkContinuation
then(List<OneTimeWorkRequest>):WorkContinua
enqueue():Operation

The WorkInfo class is a final class. A WorkInfo instance holds information about a particular WorkRequest containing the id of the WorkRequest, its current State, output, tags, and run attempt count. We can get a WorkInfo object based on the WorkRequest id from the WorkManager. We can also get a LiveData object of a WorkInfo from the WorkManager. To enable the information to be automatically updated on the UI, we can create an observer to observe this LiveData in UI controllers such as activities or fragments.

```
workManager.getWorkInfoByIdLiveData(myWork1Request.id)
.observe(this, {info->
    if (info.state.isFinished) {
        // change your UI here
        }
})
```

Test Yourself

Why use WorkManager?

It provides a unified interface for persistent background work.

Test Yourself

Briefly describe how to use WorkManager.

Conclusion

In this module, we discuss several topics. First, we show how to use broadcast receivers and intent filters. Then we explain the difference between processes and threads and introduce several different Asynchronous processing APIs, including Java thread APIs, Handler APIs and Kotlin Coroutine. Finally we discuss several types of service APIs, namely background services, foreground services, local/remote bound services, job services, and the JetPack WorkManager.