# Lecture 6

- Threads in Android
- AsyncTask
- Thread, Runnable
- Service - IntentService and Service
- Synchronized, wait, notify, notifyAll

# Threads in Android

When an application is started, the system creates a thread: the main thread. Its most important task is to manage the UI and the events. Every other component in the app is tied to this thread.

• Time-consuming tasks should be always performed on a separate thread. Otherwise, these tasks will slow down or freeze the UI. If this happens for a long time, an ANR dialog appears (Application Not Responding).

E.g: animation, data processing, heavy calculations, network communication.

• The main thread is in charge of updating the UI. Nevertheless, any other thread may call a method of a UI class using runOnUiThread:

```
activity.runOnUiThread( ARunnable  )
```

The recommended way to carry out tasks out off the UI thread is using AsyncTask.

AsyncTasks should ideally be used for short operations (a few seconds at the most.)

# AsyncTask

The class AsyncTask offers callback methods that operate off the UI thread.

```
public class Toaster extends AsyncTask<Integer,Integer,Integer> {

    protected void onPreExecute() {
        // Initialization
    }

    protected Integer doInBackground(Integer... params) {
     // Everything in this method is done in a separate thread.
     // Calling publishProgress (Integer ...); will result on a call to
     // OnProgressUpdate (Integer ...), which will notify the UI thread of
     // any message
        return result;
    }


    protected void onProgressUpdate(Integer... update) {
        // Publish results to the UI thread
    }

    protected void onPostExecute(Integer result) {
        // Called when the thread has finished
    }
}
```

Start the AsyncTask with:

```
Toaster toaster = new Toaster();
toaster.execute( integer-values );
```

# Thread

Thread class represents a thread in java.

**Methods:**

- start(): Starts the execution of the run method. Start() method should only be called once per thread.
- interrupt(): Sets a flag that the thread should be discontinued. If the thread is blocked, an InterruptedException is launched.
- setPriority(int prioritet): Changes the priority of the thread
- interrupted(): Returns true if interrupt has been called.
- join(): The program waits until the thread is finished
- yield(): Pauses the thread, so other threads (with the same priority) can be executed.

**Class methods**

- sleep(int ms): The thread waits ms milliseconds
- currentThread(): Returns a reference to the thread.

# The thread life cycle

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated ( Dead ):** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

# Thread extends Thread

In java create new topics with the class Thread. You can use one of the following ways:

1. By inheriting class Thread. Get an instance, call start on it and override run.

```
public class ClassA extends Thread {
   :
   public void run() {
      // Code here what the thread must do
   }
}
----------------------------------------------------
// Instantiate and launch the thread
ClassA c = new ClassA
c.start();
```

# Thread implements Runnable

2. A class that implement the interface Runnable, and a Thread object.

```
public interface Runnable {
   public abstract void run();
}

public class ClassB implements Runnable {
   :
   public void run() {
      // Code here what the thread must do
   }
}
-----------------------------------------------------------
// Instantiate and launch the thread
Thread thread = new Thread(new ClassB());
thread.start();
```

# Service

A Service is a component executed in the background.

- A Service can perform long-running operations in the background and does not provide a user interface.
- Another application component can start a service and it will continue to run in the background even if the user switches to another application.
- Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC).
- For example, a service might:
    - handle network transactions,
    - play music,
    - perform file I/O,
    - or interact with a content provider, all from the background.

# Service

A service can essentially take two forms:

**Started**

A service is "started" when an application component (such as an activity) starts it by calling startService(). Once started, a service can run in the background indefinitely, even if the component that started it is destroyed. Usually, a started service performs a single operation and does not return a result to the caller. For example, it might download or upload a file over the network. When the operation is done, the service should stop itself.

**Bound**

A service is "bound" when an application component binds to it by calling bindService(). A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC). A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed.

# Service

A service should be declared in AndroidManifest.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="se.mah.tsroax.f7intentservice" >

   <application
      :
      <activity …>
         :
      </activity>

      <service
         android:name=".ServiceA"
         android:enabled="true”                <!– can be instantiated -->
         android:exported="false" >            <!– only started by this app -->
      </service>
   </application>

</manifest>
```

# IntentService

The class **IntentService**:

- It is a simple way to implement a Service.
- Automatically starts a separate thread.
- The Intent (the app) is placed in a buffer. All Intents in the buffer are performed sequentially.
- It turns itself off when the buffer is empty. Starts up itself if a new Intent comes into the buffer.

A class that inherits **IntentService** shall include

- A constructor without parameters
- The method onHandleIntent

```
public class ServiceA extends IntentService {
    public ServiceA() {
        super("ServiceA");
    }

    protected void onHandleIntent(Intent intent) {
  // Handles the Intent
    }
}
```

# IntentService - examples

```
public class ServiceA extends IntentService {
    private boolean serviceRunning;

    public ServiceA() {
        super("ServiceA");
    }

    public int onStartCommand(Intent intent, int flags, int startId) {
        Log.d("onStartCommand",(intent==null)?"Null":"Not null");
        serviceRunning = true;
        return super.onStartCommand(intent,flags,startId);
    }

    public void onDestroy() {
        Log.d("onDestroy","Service down");
        serviceRunning = false;
        super.onDestroy();
    }

    protected void onHandleIntent(Intent intent) {
        Log.d("onHandleIntent","begin");
        int times = intent.getIntExtra("times", 0);
        for(int i=0; i<times && serviceRunning; i++) {
            Log.d("onHandleIntent", "Nbr " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {}
        }
        Log.d("onHandleIntent","end");
    }
}
```

# Service

Class Service:

- More code to write than just using IntentService.
- Must use at least one separate thread (explicitly).
- Must handle repeated calls from the application.
- Can be stopped by the application or by itself.

A class that inherits **Service** shall include:

- The method onStartCommand
- The method onBind which will return null (so far)

# Service

A service that starts a new thread for each call to start might look like this:

```
public class ServiceA extends Service {
    private boolean serviceRunning;

    public void onCreate() {
        super.onCreate();
        serviceRunning = true;
    }

    public int onStartCommand(Intent intent, int flags, int startId) {
        if(intent!=null) {
            Thread thread = new Thread(new MyLogger(intent));
            thread.start();
        }
        return Service.START_STICKY;
    }

    public void onDestroy() {
        serviceRunning = false;
        super.onDestroy();
    }

    // See next slide
}
```

# Service

Cont:

```
public class ServiceA extends Service {

    private class MyLogger implements Runnable {
        private Intent intent;

        public MyLogger(Intent intent) {
            this.intent = intent;
        }

        public void run() {
            int times = intent.getIntExtra("times", 0);
            for (int i = 0; i < times && serviceRunning; i++) {
                Log.d("MyLogger - run","Nbr " + i);
                try {
                    Thread.sleep(3000);
                } catch(InterruptedException e) {}
            }
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

# Synchronized methods

If multiple threads are using the same resource, you have to ensure that only one thread is using it at the same time. Specially when those threads change the content in the resource.

You do one of the following ways:
• Specify that certain methods should be synchronized.

```
public synchronized int setTal () {
  :
}
```

When a thread calls the object method setTal, the system prevents other threads to call this method until is execution is finished.

# Synchronized blocks

• You can synchronize a block of code with the synchronized statement. The block can’t be accessed by other threads until the first thread that entered it has passed though.

```
synchronized (this) {
  :
}
```

# Wait, notify, notifyAll

These method existing in Object provide some useful functionality for threads. They can only be called from synchronized methods or synchronized blocks.

- public void wait () public void wait (long ms)

The thread is interrupted (wait ()) or stopped for a given time (wait (long)). The thread can be resumed by calling notify () / notifyAll ().

- public void notify ()

An interrupted thread is resumed.

- public void notifyAll ()

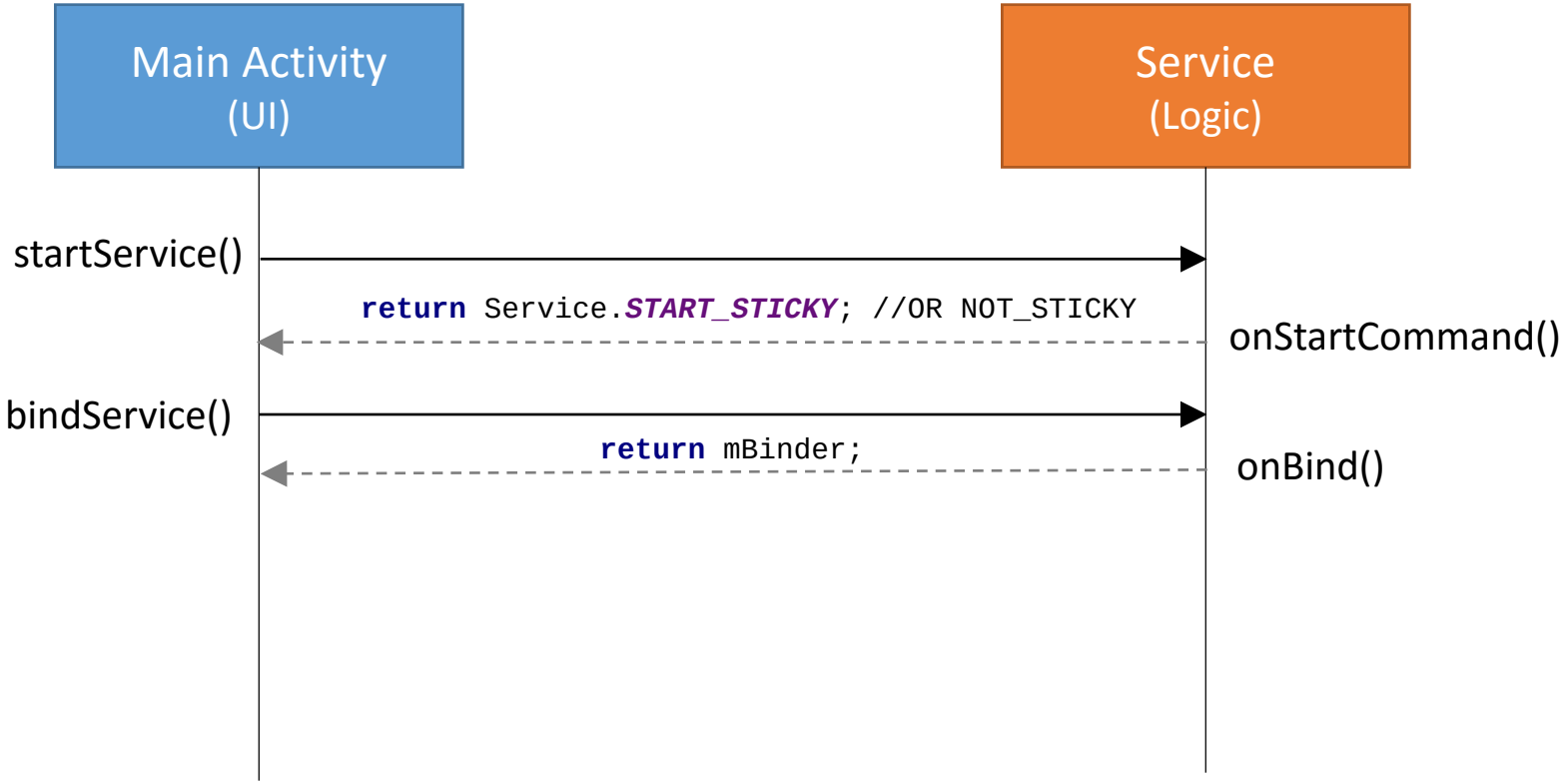
All interrupted threads are resumed.

# BONUS TRACK: Launching and binding to services

- Services are started by Activities.
- There are mainly two ways for starting services (depending on your needs):

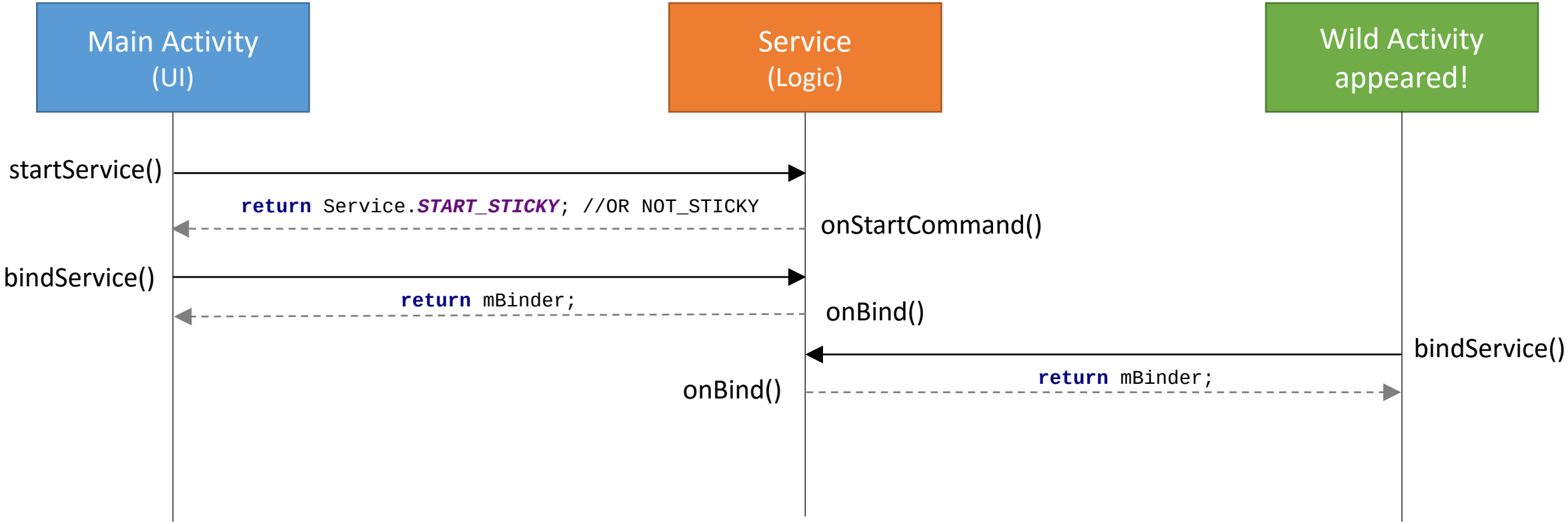
# BONUS TRACK: Launching and binding to services

1. Start a service that will be operating until you explicitly stop it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - Even if all bounded activities unbind, the service will be still running.
   - The service only stops by calling stopSelf (from the service) or stopService (from outside).



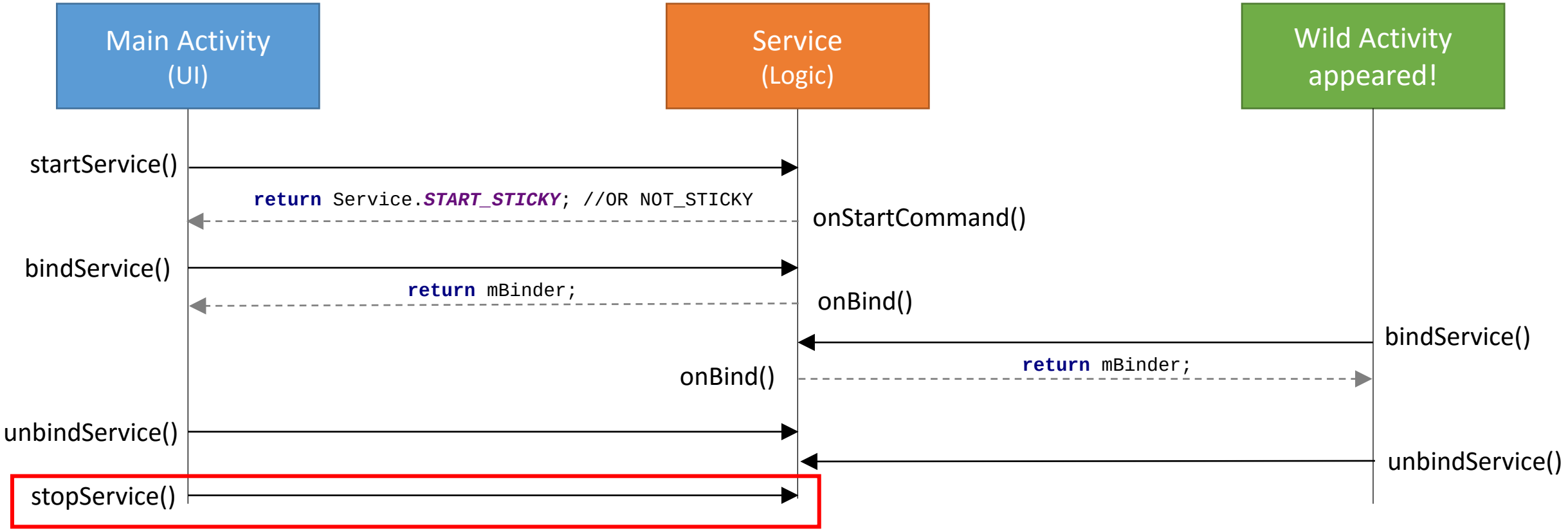
# BONUS TRACK: Launching and binding to services

1. Start a service that will be operating until you explicitly stop it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - Even if all bounded activities unbind, the service will be still running.
   - The service only stops by calling stopSelf (from the service) or stopService (from outside).



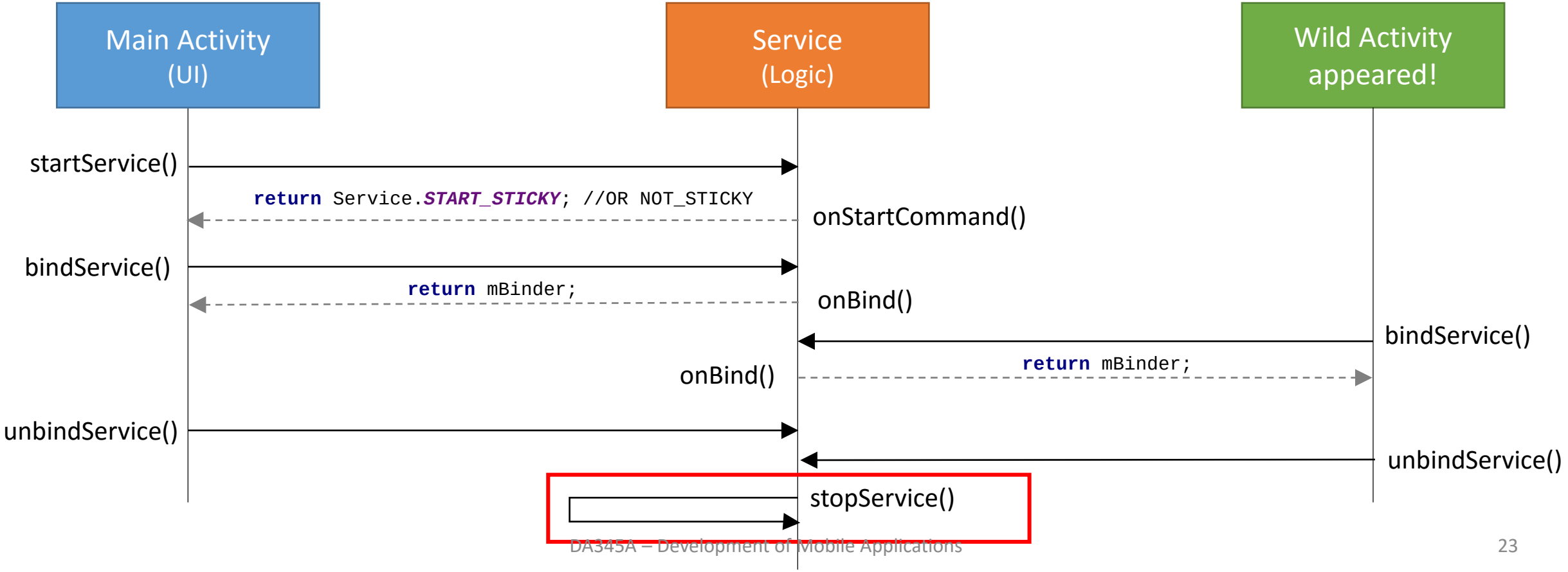
# BONUS TRACK: Launching and binding to services

1. Start a service that will be operating until you explicitly stop it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - Even if all bounded activities unbind, the service will be still running.
   - The service only stops by calling stopSelf (from the service) or stopService (from outside).



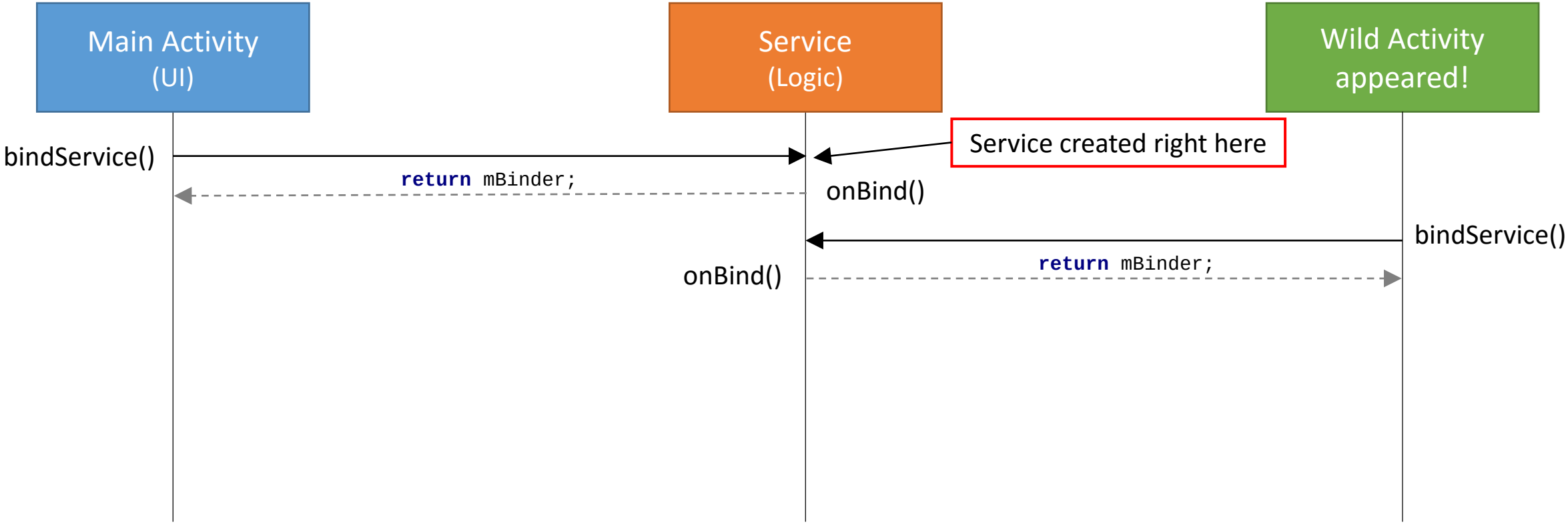
# BONUS TRACK: Launching and binding to services

1. Start a service that will be operating until you explicitly stop it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - Even if all bounded activities unbind, the service will be still running.
   - The service only stops by calling stopSelf (from the service) or stopService (from outside).



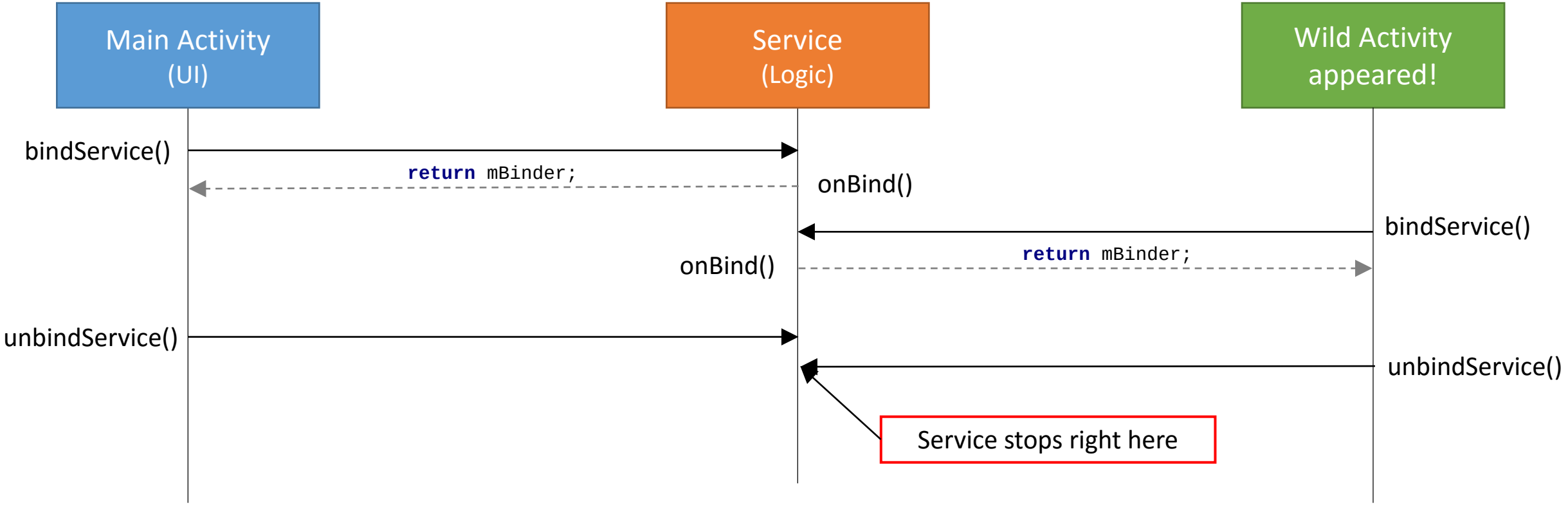
# BONUS TRACK: Launching and binding to services

2. Start a service by binding to it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - When all bounded activities unbind, the service will stop running.
   - **You never call neither startService nor stopService/stopSelf.**



# BONUS TRACK: Launching and binding to services

2. Start a service by binding to it.
   - Activities will be able to bind/unbind to it (bindService()/unbindService()).
   - When all bounded activities unbind, the service will stop running.
   - **You never call neither startService nor stopService/stopSelf.**



# BONUS TRACK: Detecting running services

- Whenever you need to check whether a given service is running or not.
- **Retrieve an instance of the ActivityManager.**
- It will provide a list with all the running services by calling getRunningServices()

```
private boolean isMyServiceRunning(Class<?> serviceClass) {
    ActivityManager manager = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);
    for (RunningServiceInfo service : manager.getRunningServices(Integer.MAX_VALUE)) {
        if (serviceClass.getName().equals(service.service.getClassName())) {
            return true;
        }
    }
    return false;
}
```