Lesson 9. Android's Concurrency

Android's Concurrency Mechanisms 3

The importance of multitasking. 3

Multitasking: Pros and Cons 4

Tools for Implementing Concurrency Control 4

Threads and Runnables 5

Monitors 6

ReadWriteLocks 7

Counting Semaphores 7

AtomicTypes 8

BlockingQueues 8

Example 9.1 Classic Java Threads and Runnables 9

Android's Handler Class 12

Example 9.2 Android's Handler Class – Passing Messages 14

Example 9.3 Android's Handler Class – Posting 19

The AsyncTask Class 22

Example 9.4 Using the AsyncTask class 24

Android Internal Services 27

Starting and stopping Services 28

Service Life-Cycle 28

Life-Cycle of a Standard Service 29

Life-Cycle of an IntentService 29

Broadcast Receivers 30

Example 9.5 Using the IntentService class 31

Notifications 38

Example 9.6 Using the Standard Service class 40

Lesson 9

Android's Concurrency Mechanisms

Goals

In this lesson, we study different ways in which Android supports concurrency. This concept is important because it provides the foundation for applications to perform many tasks at the same time in a transparent way. First, we will review classic Java strategies for managing access to critical regions. Then, we will explore new tools such as the native Android `Handler` and `AsyncTask` classes. Finally, we will discuss Android's Background Services and Broadcast Receivers.

The importance of multitasking.

In the early days of computing, mainframe machines had only one CPU to do all their work. Early operating systems were not able to process more than a single task at the time. The idea of eliminating "CPU's idle-time" propelled the software community to come with ideas about how to maximize the use of the precious CPU resource. New multitasking operating systems emerged giving room to strategies and tools such as Time-slicing, Time-sharing, interruption-management, monitors, semaphores, and so on. Meanwhile, the hardware community has maintained a continuous state of innovation resulting in more computing power hosted by smaller, cheaper, more portable devices. Today, most computers - and smartphones, in particular- include an ever-growing number of powerful processors, as well as an array of dedicated controls to manage their embedded hardware units including video, audio, communications, and even medical components. The goal of *multitasking* is perhaps, the single-most important reason for all these revolutionary changes to occur.

Multitasking occurs when an application is simultaneously running multiple cooperative operations, which are part of the same logical unit. For instance, think of an Android app that shows an animation while is downloading a large file from a website, and in spite of all this work, maintains a responsive UI for the user to enter data. The implementation of such an app is more involved than the single-activity kind of projects that we have tackled before. One possible solution strategy is to allow the main-activity to split and allocated those individual concurrent actions in separate **threads**.

A thread is the smallest unit of computing that can be controlled by the operating system. That is, when supervising a thread, the OS scheduler knows whether the thread is ready, waiting, sleeping, running, blocked, or dead. The scheduler may also decide what to do with the thread based on signals sent to it such as start(), stop(), wait(), etc. A thread is relatively independent of other work happening around it. Observe that a thread may continue to work indefinitely, even after the termination of the activity that initiated its life cycle. Each thread has its private memory space, processor resources, and call **stack** which is used for method calls, parameter passing, and storage of the called method's local variables. Figure 9.1 depicts a state diagram rendition of Java threading.

At the heart of Android, there is Java and Linux. We know that the Java Virtual-Machine (JVM) provides its own **Multi-Threading** architecture; consequently, the JVM and the Android Run-Time Environment are hardware independence. Each Virtual Machine instance has at least one **main thread**. Threads in the same

Virtual Machine interact and synchronize by the use of **shared objects** and **monitors**. We will review classic (and new) Java synchronization artifacts in the next section. Again, this is important, because Android apps run –in addition to its native mechanisms- the same virtual multitasking scheme of Java.

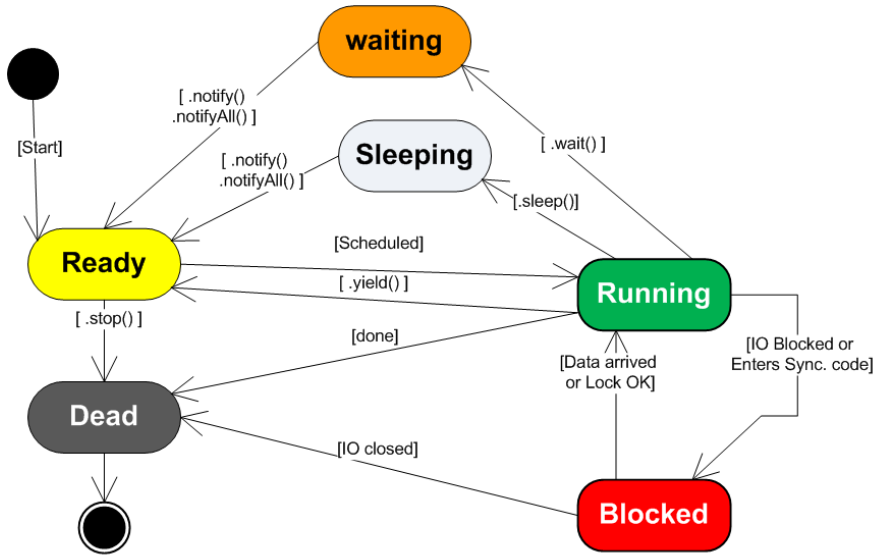


Figure 9.1 Machine independent life-cycle of a Java Thread showing states and signals

Multitasking: Pros and Cons

(+) Scheduling Abstraction. An app's execution pattern could be drawn based on whether its logical components execute in a serial or parallel fashion. Those logical pieces that must run one after the other are better placed in an activity class, while those that make sense running in parallel are encapsulated inside independent threads.

(+) Data Sharing. Threads could *read-and-write* any of the data resources held in the process that contains them.

(+) Responsiveness. App's logic can be classified into two broad categories: (a) quick and responsive, and (b) slow Data-CPU bound. The quick-type portion of logic belongs in the main application's thread where the UI is managed, while slow processes can be assigned to background threads.

(+) Hardware Support. A multithreaded program operates faster on computer systems that have multiple CPUs. Observe that most current Android devices do provide multiple processors.

(-) Overhead. Multitasking code tends to be more complex**.** Parallel computing is not trivial; therefore, developing, debugging, and maintaining this kind of programs requires more effort.

(-) Run-Time Vigilance. You may need to avoid or detect-and-resolve **deadlocks.**

Tools for Implementing Concurrency Control

The goal of **concurrency control** is to avoid, detect, and resolve **conflicts** that may occur with processes that simultaneously access and alter **shared resources** (also called **critical regions,** or **critical resources**). Data **inconsistency** is the perhaps the most severe problem arising from the uncontrolled sharing of data. This

anomaly commonly occurs when a process attempts to read a data item at the same time that another activity is trying to update that value. If left alone, the action of concurrent activities on a global data resource may lead to unpredictable results. Note that relying on a simplistic scheme granting exclusive access to data items may generate excessive deadlocks.

Protected sharing of critical regions is a complex process that requires the programmer's participation and the vigilant supervision of the Operating System. Assume the intention of a process when accessing a critical resource is either to observe the resource (we call it a **reader)** or to modify the critical region **(**we call it a **writer.)** The safe sharing protocol is enforced by a policy in which

- A process is allowed to read an item provided that it is not already given to a writer. Other readers are welcome to access the data simultaneously. If a process wants to write to the item, it must wait for the completion of all readers.
- A process must receive exclusive permission to write to a resource. Other processes wanting to read or write to the same data item must wait for the completion of the writer's action.

Let us review some of the software mechanisms supplied by Java to implement and control concurrency.

Threads and Runnables

`Threads` and `Runnables` are Java classes implementing parallel work. Each Android activity runs in a thread usually called the "main activity thread". The main thread manages the UI and in most cases, it is enough for hosting the entire app's code. However, there are occasions in which some actions should be executed in parallel. In those cases, the activity's logic is decomposed in several programming components. The main thread takes care of the UI, and any other parallel logic is framed inside a concurrent container such as Java `Thread` or a `Runnable` class.

Each thread is an independent and self-sufficient environment, with its calling stack, memory map, and processor resources. A thread can define its local variables and methods and has access to all the data items defined in the main thread. Here is where caution is needed. The programmer should make efforts in using controlled access of those shared data elements using tools such as semaphores, monitors, AtomicTypes, etc. (more on this topic in a moment.) The following are two strategies for creating and executing a Java Thread.

Style1. Create a new `Thread` instance passing to it a `Runnable` object.

```
Runnable myRunnableObj1 = new MyRunnable();
Thread t1 = new Thread(myRunnableObj1);
t1.start();
```

where the runnable class is defined as

```
class MyRunnable implements Runnable {
   //Local variables here...
   public MyRunnable() {
      //some initialization - if needed!
   }
   public void run() {
      // your parallel logic here...
      }
}
```

Style2. Create a new custom sub-class that extends `Thread` and override its `run()` method.

```
MyThread t2 = new MyThread();
t2.start();
```

In both cases, the **start()** method must be called to execute the new Thread. For a complete sample, see Example 9.1.

> **Warning** In Android, the main activity is responsible for the UI's management. The duty of this thread is to maintain a responsive interface that quickly responds to user interactions. Any slow logic that blocks the UI belongs in a parallel thread; otherwise, Android issues the dreaded exception "Application Not Responding" (**ANR**).
> **Only the main thread can update the UI.** Any parallel thread attempting to change the UI or Toast a message will raise a fatal run-time exception.
> Background threads can read any of the main's variables. They also can update any shared variable, as long as it is not a UI element.

Monitors

A monitor (or Mutex) is a region of critical code executed by only one thread at the time. In a Java implementation of a monitor, you may use the **synchronized** modifier to acquire a mutually **exclusive lock** on an object (data or code). If an object is exclusively locked, other threads must wait until the lock on that object is released. The following fragment illustrates a simple strategy to implement a monitor protecting a region of critical code (synchronized statement).

```
synchronized (object){
   // place here your exclusive code
   // only one thread will work at the time!
}
```

The next code fragment shows how a monitor is used to protect the integrity of a shared data item called `globalVar`. Assume that only operations such as *get*, *set*, and *add* are applied on the item. Performing those actions becomes a matter of policy. The provided synchronized methods `getGlobalVar(), setGlobalVar`, and `IncreaseGlobalVar()` **should** be used to accomplish our goal. However, nothing prevents a rogue thread to bypass the monitor and perform a non-safe access or mutation on the variable, for instance, `globalVar++`.

```
int globalVar = 0;
...
public synchronized  void methodToBeMonitored() {
   // place here your code to be lock-protected
   // (only one thread at the time!)
}

public synchronized int getGlobalVar() {
   return globalVar;
}

public synchronized void setGlobalVar(int newGlobalVar) {
   this.globalVar = newGlobalVar;
}

public synchronized int increaseGlobalVar(int inc) {
   return globalVar += inc;
}
```

> **Warning** **synchronized** doesn't support separate locks for reading and writing. Access based on

mutual exclusion is inherently slow (only one thread at the time is allowed). Furthermore, mutex protocol is prompt to deadlocks. Generally, `ReadWriteLocks` offer a better solution.

ReadWriteLocks

In general, you should expect better performance from a set of concurrent processes when multiple threads are allowed to read from a shared resource simultaneously. Still, only one writer can enter the critical region at the time. Java supports dual Read/Write locks as shown below

```
ReadWriteLock rwLock = new ReentrantReadWriteLock();

   rwLock.readLock().lock();
   // multiple readers can enter this section
   // (as long as no writer has acquired the lock)
   rwLock.readLock().unlock();

   rwLock.writeLock().lock();
   // only one writer can enter this section,
   // (as long as no current readers locking)
   rwLock.writeLock().unlock();
```

Counting Semaphores

Counting Semaphores are useful in situations in which a specific number of shared objects needs to be safely shared among threads demanding (and releasing) a variable number of those resources. In the fragment below, a semaphore reserves up to *n* permits. A thread trying to enter the critical section will first try to acquire *n1* of the remaining passes. If all of the *n1* tokens are obtained it enters the critical section, does it work, and then release *n2* passes ($n2 \leq n1$). If all requested passes cannot be obtained, the thread waits in the semaphore until they become available. (Caution: starvation may occur, seniority rights are not preserved.)

```
int n = 10;
try {
   Semaphore semaphore = new Semaphore(n);

   semaphore.acquire(n1);
   // put your critical code here
   semaphore.release(n2);

} catch (InterruptedException e) {
   // take care of the problem here
}
```

Warning	**Semaphores** are powerful tools for modeling gatekeeping scenarios. However, they have two potential drawbacks: *starvation* and *loss of seniority rights*. There is no guarantee that a particular process waiting for the critical region is ever allowed into it. Similarly, the longest waiting thread is not necessarily the next in entering the critical region once resources become available.

AtomicTypes

Atomic types provide a set of t**hread-safe** classes whose objects could be fetched and modified in a single indivisible (atomic) operation. Consider the following code fragment

```
Integer a = 1; ...; a++;
```

- The **a++** increment operation is in reality made of three steps: (1) fetch the value a, (2) add 1 to a, and (3) write the new value back to a.
- An **atomic execution** of a++ would require the three steps above to behave as a single one-step operation.

Java atomic types include among others

```
AtomicInteger, AtomicLong, AtomicDouble, AtomicFloat, AtomicString, AtomicFile,
AtomicIntegerArray, ...
```

AtomicTypes include support for indivisible methods such as:

```
.get()   .getAndSet(),  .getAndIncrement(), .getAndDecrement(), .getAndAdd(),
.set()   .compareAndSet()
```

BlockingQueues

The **BlockingQueue** class exposes a synchronized queue to any number of producers and consumers. It is implemented using one of the following concrete classes: `ArrayBlockingQueue, DelayQueue, LinkedBlockingDeque, PriorityBlockingQueue`, and `SynchronousQueue`. Adding and removing messages from the queue are thread-safe operations. Consumers are moved to a wait state when the queue is empty and are awoken when new entries become available. A producer must gain exclusive rights to append a new token to the queue. While one producer is working, other producers wanting to add tokens must wait.

The fragment below shows a typical usage of a `BlockingQueue`. The classes `Producer` and `Consumer` are defined as `Runnables`. The queue is managed through atomic operator such as `.put(), .take(), .removeAll(), .peek(), .size(), .clear(),` etc.

```
ArrayBlockingQueue<String> queue = new ArrayBlockingQueue<String>(4);
Producer producer = new Producer(queue);
Consumer consumer = new Consumer(queue);
new Thread(producer).start();
new Thread(consumer).start();
```

Producer(s)	BlockingQueue	Consumer(s)
Producer1 queue.put("Data1A") Producer2 queue.put("Data2A")		token = queue.take() Consumer1 token = queue.take() ConsumerX

Example 9.1 Classic Java Threads and Runnables

This example implements a simple app consisting of the main UI thread and two parallel threads. The app's UI shows two buttons to stop the background tasks respectively. Each of the parallel threads uses the statement `Thread.sleep(1000)` to create an artificial delay similar to a burst of busy work (1000 milliseconds). The back-workers employ the Android-Studio **LogCat** mechanism to report their progress.

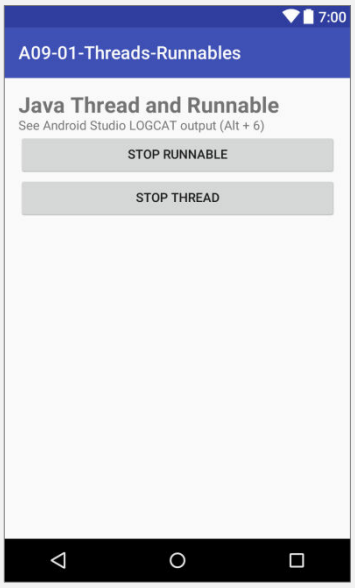
Assembling App 9.1

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-01-Threads-Runners`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Modify the app's main layout (`res\layout\activity_main.xml`) as follows.

Example 9.1 Layout definition: activity_main.xml (A09-01-Threads-Runners)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    android:id="@+id/content_main"
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Java Thread and Runnable"
        android:textAlignment="center"
        android:textSize="24sp"
        android:textStyle="bold"/>

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="See Android Studio LOGCAT output (Alt + 6)"
        android:textAlignment="center"/>

    <Button
        android:id="@+id/btnStopThread1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Stop Runnable"/>

    <Button
        android:id="@+id/btnStopThread2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Stop Thread"/>
```



```
</LinearLayout>
```

Step 3. Modify your Activity's code. State the main class with the following code. For simplicity, we have defined both parallel threads as embedded classes. See Figure 9.2 for a sample of the `LogCat` created during an execution of this app.

Example 9.1 MainActivity.java (A09-01-Threads-Runners)

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {

   String LOG_TAG = "<<A09-01-Threads>>";
   Button btnStopThread1, btnStopThread2;
   public volatile AtomicBoolean thread1MustDie = new AtomicBoolean(false);                    (1)
   public volatile AtomicBoolean thread2MustDie = new AtomicBoolean(false);

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      //UI plumbing
      setContentView(R.layout.activity_main);
      btnStopThread1 = (Button) findViewById(R.id.btnStopThread1);                             (2)
      btnStopThread2 = (Button) findViewById(R.id.btnStopThread2);
      btnStopThread1.setOnClickListener(this);
      btnStopThread2.setOnClickListener(this);
      // Thread1 runs a custom Runnable
      Thread t1 = new Thread(new MyRunnable() );                                               (3)
      t1.start();
      // Thread2 overrides its own run method (uses no runnable)
      MyThread t2 = new MyThread();
      t2.start();
   }//onCreate

   public class MyRunnable implements Runnable {                                              (4)
      @Override
      public void run() {
         try {
            for (int i = 1000; i <= 2000 && !thread1MustDie.get() ; i++){
               Thread.sleep(1000);
               Log.e (LOG_TAG, "t1 - [runnable] ..." + i);
            }
         } catch (InterruptedException e) {
            Log.e (LOG_TAG, "t1 - [runnable] ..." +  e.getMessage() );
         }
      }//run
   }//class

   public class MyThread extends Thread{                                                      (5)
      @Override
      public void run() {
         super.run();
         try {
            for(int i=1; i<=100 && !thread2MustDie.get() ; i++){
               Thread.sleep(1000);
```

```
                Log.e (LOG_TAG, "t2 - [thread] ..." + i);
            }
        } catch (InterruptedException e) {
            Log.e (LOG_TAG, "t2 - [thread] ..." + e.getMessage() );
        }
    }//run
}//MyThread

@Override
public void onClick(View view) {                                                    6
    if(view.getId() == btnStopThread1.getId()){ thread1MustDie.set(true); }
    else if (view.getId() == btnStopThread2.getId()){ thread2MustDie.set(true); }
}//onClick

@Override
protected void onDestroy() {                                                        7
    //make sure back-workers stop when the main activity is gone
    super.onDestroy();
    thread1MustDie.set(true);
    thread2MustDie.set(true);
    Log.e(LOG_TAG, "All done!");
}
}
```

Comments

1. We define two thread-safe control variables. When any of those variables changes to `true`, its associated thread will stop. The modifiers `volatile AtomicBoolean` guarantee that threads accessing this data item will see the most updated version of it. Also, any updates to the variables will be performed as an atomic operation.
2. The main thread performs routine UI plumbing work. When a button is clicked the app must immediately react to the request, and change the control variable. The associated thread will at some point realize the new value and terminate.
3. First, a conventional Java thread `t1` is created and started. It receives an instance of the custom made `MyRunnable` class where all its logic is contained. Then, we make a thread `t2` from a custom class that extends the Java `Thread` class.
4. The `run()` method inside the Runnable class *fakes* the execution of some slow work. The method uses a `volatile AtomicBoolean` variable to short-circuit its execution.
5. The `run()` method inside the extended `Thread` class *fakes* the execution of some slow work. The method uses a `volatile AtomicBoolean` variable to short-circuit its execution.
6. Clicking any of the app's buttons ("Stop Thread", "Stop Runnable") safely changes its corresponding control variable. Note the use of the `var.set(true)` method.
7. When the main thread ends, the `onDestroy()` method is called. There we make sure both control variables are changed, so the background tasks will be able to terminate simultaneously. Note that background threads may continue to work, even after the termination of the activity that triggered them.

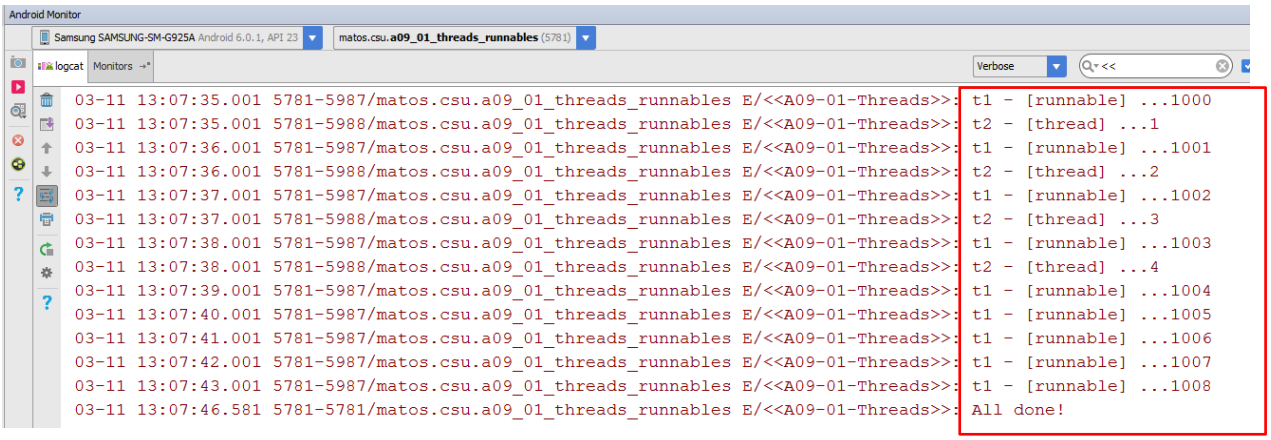


Figure 9.2 Example 9.1's sample of execution.

Android's Handler Class

There are occasions in which the pieces of an application must be decoupled into heavy-load and quick-responsive tasks. The heavy-load tasks usually carry out demanding computations or interact with slow data sources and communication channels. On the other hand, the quick-responsive tasks are typically associated with controlling the application's GUI so it could continue to interact effectively with the user during periods of heavy load. Android offers two structural ways for dealing with this light/heavy-duty computing scenario, you either

(1) Perform expensive operations in a background *service*, using *notifications* and *receivers* to inform users about progress made, or

(2) Do the slow work inside a *background thread* or an *AsyncTask* object.

Android threads can share memory using any of the Java thread-safe mechanisms (locks, semaphores, queues, etc.) However, background workers are not allowed to interact with the GUI. Only the main process can access the activity's view and interact with the user.

In this context, Android's main task behaves like a consumer and the background threads as producers. The main thread is responsible for defining a `Handler` object and optional `Runnable` objects. The main activity may use its *MessageQueue* to manage interactions between the main and the background threads it creates. The message queue acts as a `BlockingQueue` with the capacity to enqueue tokens containing messages or runnables sent by the secondary threads. By protocol, children threads must request empty tokens from the ancestor's queue, fill them up, and then send back to the parent's queue. Those requests will be executed in the order in which they are removed from the message queue. A more detailed description of the process follows.

- A *background-to-foreground* thread communication is initiated by the background worker (producer) by requesting a message token from the main thread (consumer). The `obtainMessage()` method is used to negotiate the acquisition of the token, which acts as a special envelope with various pre-defined compartments for data to be inserted.
- After an empty token is received, the background thread can enter its local data into the message token. Local data could be anything ranging from a few numeric values to any custom object. Finally the token is attached to the Handler's message queue using the `sendMessage()` method.

- The consumer's Handler uses the `handleMessage()` method to listen for new messages arriving from the producers. A message taken from the queue to be serviced, could either pass some data to the main activity or request the execution of `Runnable` objects through the `post()` method.

Figure 9.3 illustrates a situation in which the main activity communicates with two background threads. One of them uses the handler's queue to pass messages to the main activity. Those messages could be used to update the activity's GUI if needed. The second worker posts some foreground thread to be executed and perform some action on behalf of the background thread.

A typical `Message` object contains various predefined fields **[what, arg1, arg2, obj]** that are commonly used to store quickly simple data values (*arg1, arg2*), identify the calling thread (*what*), and supply an arbitrary object (*obj*). Table 10.1 shows the layout of a Handler's message object and gives a list of methods commonly used by handlers.

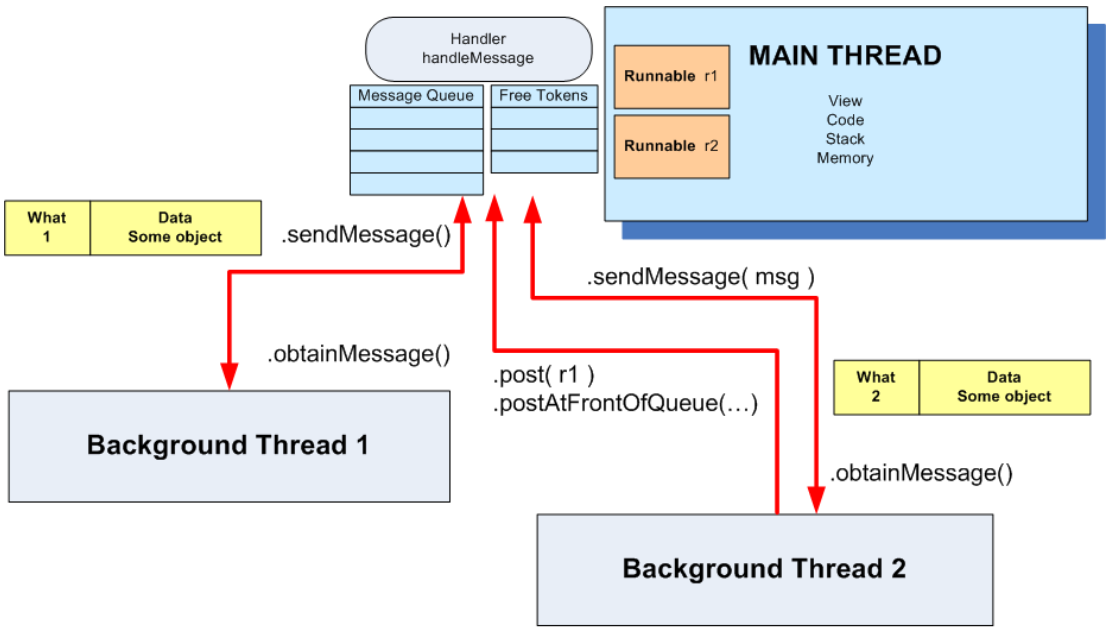


Figure 9.3 Android back-threads using a `Handler` to communicate with the main activity.

Note Commonly, a Handler's listener is set in the main activity to wait for messages sent by background tasks. However, each worker thread could also define its local handler. Consequently, a worker thread could receive messages from other threads, including main.

Returns	Method / Description

	what (int) | **arg1** (int) | **arg2** (int) | **obj** (any type of object) Layout of a Handler's Message object.
`Void`	`handleMessage(Message msg)` Handler subclasses must implement this method to receive messages.
`final Message`	`obtainMessage()` Returns a new `Message` from the global message pool.
`final Message`	`obtainMessage(int what, int arg1, int arg2, Object obj)` Obtains a free token from the message pool. You assign values to what, obj, arg1,and arg2.
`final boolean`	`post(Runnable r)` Adds Runnable `r` to the message queue.
`final boolean`	`sendMessage(Message msg)` Pushes a message onto the end of the message queue.
`final boolean`	`sendMessageAtFrontOfQueue(Message msg)` Inserts a priority message at the front of the message queue.
`boolean`	`sendMessageAtTime(Message msg, long uptimeMillis)` Enqueues a message into the message queue before the absolute time uptimeMillis.
`final boolean`	`sendMessageDelayed(Message msg, long delayMillis)` Enqueues a message into the queue at time (current time + delayMillis).

Table 10.1 Selected methods of the Handler class

The next two examples detail how a background thread uses the message queue to communicate with the main activity. The first case (Example 9.2) illustrates the message passing strategy, and then we explore the posting of runnables (Example 9.3).

Example 9.2 Android's Handler Class – Passing Messages

This example implements a simple app consisting of its main UI activity and a parallel thread performing some slow operation. The app *must* be organized as two simultaneous tasks. Otherwise, the slow component would have frozen the UI making the app non-responsive for periods of time. Observe that apps exposing a UI with a slow user-reaction time are unfit for the Android platform. Such apps are branded as ANR (application not responding) and are terminated by the Operating System.

The app's UI shows a button. When pressing; it immediately displays the current date on the screen (as evidence of prompt responsiveness.) The background thread periodically inserts messages in the Handlers

queue. The Handler's `messageHandle()` method executes each time that a token is removed from the main's queue.

Assembling App 9.2

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-02-HandlerMessages`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Modify the app's main layout (`res\layout\activity_main.xml`) as follows. The UI includes a button that when pushed will immediately display the current date and time. There is a circular and a linear progress bar to indicate the current state of the task performed by the back worker. A scrolling view shows each message received by the main thread from the parallel thread.

Example 9.2 Layout definition: activity_main.xml (A09-02-HandlerMessages)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:orientation="vertical"
              android:padding="16dp">

    <Button
        android:id="@+id/btnWhatTime"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Push me any time!"/>

    <TextView
        android:id="@+id/txtCurrentTime"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="#ffff00"
        android:gravity="center"
        android:text="txtCurrentTime"/>

    <TextView
        android:id="@+id/txtWorkProgress"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:padding="10dp"
        android:text="Working ...."
        android:textSize="18sp"
        android:textStyle="bold"/>

    <ProgressBar
        android:id="@+id/progress1"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"/>

    <ProgressBar
        android:id="@+id/progress2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"/>
```


```
    <ScrollView
        android:id="@+id/myscroller"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <TextView
            android:id="@+id/txtReturnedValues"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_margin="7dp"
            android:background="#2962ff"
            android:padding="4dp"
            android:text="returned from thread..."
            android:textColor="@android:color/white"
            android:textSize="14sp"/>
    </ScrollView>

</LinearLayout>
```

Step 3. Enter your app's code. Use the following code to establish the app's `MainActivity` class. Observe that in this example the main thread, Handler, and background thread are all defined in the `MainActivity`.

Example 9.2 MainActivity.java class (A09-02-HandlerMessages)

```
public class MainActivity extends AppCompatActivity {
   ProgressBar barHorizontal; //linear
   ProgressBar barCircular; //circular
   TextView txtCurrentTime;
   TextView txtWorkProgress;
   TextView txtReturned;
   ScrollView myScrollView;
   Button btnWhatTime;
   // this is a control var used by backg. threads
   volatile AtomicBoolean isRunning = new AtomicBoolean(false);
   // lifetime (in seconds) for background thread
   final int MAX_SEC = 20;
   // protected variable accessed by the background thread
   volatile AtomicInteger mainGlobalVar = new AtomicInteger(0);

   Handler handler = new Handler() {                                    1
      @Override
      public void handleMessage(Message msg) {
         // extract incoming object
         String returnedValue = (String) msg.obj;
         // show the value sent by the background thread here
         txtReturned.append("\n" + returnedValue);
         myScrollView.fullScroll(View.FOCUS_DOWN);
         // update the progress bar
         barHorizontal.setMax(MAX_SEC);
         barHorizontal.incrementProgressBy(1);
         // testing early termination
         if (barHorizontal.getProgress() == MAX_SEC) {
            txtReturned.append(" \nDone \n Back thread has finished ...");
```

```
            isRunning.set(false);
        }

        if (barHorizontal.getProgress() == barHorizontal.getMax()) {
            txtWorkProgress.setText("Done");
            barHorizontal.setVisibility(View.INVISIBLE);
            barCircular.setVisibility(View.INVISIBLE);
        } else {
            txtWorkProgress.setText( "Variable mainGlobalVar ... " + mainGlobalVar
                                + "\nProgress... " + barHorizontal.getProgress() + "/" + MAX_SEC) ;
        }
    }
}; // handler

@Override
public void onCreate(Bundle icicle) {
    super.onCreate(icicle);
    setContentView(R.layout.activity_main);
    // UI plumbing
    barHorizontal = (ProgressBar) findViewById(R.id.progress1);                    2
    barHorizontal.setProgress(0);
    barHorizontal.setMax(MAX_SEC);
    barCircular = (ProgressBar) findViewById(R.id.progress2);
    txtWorkProgress = (TextView) findViewById(R.id.txtWorkProgress);
    txtReturned = (TextView) findViewById(R.id.txtReturnedValues);
    txtCurrentTime = (TextView)findViewById(R.id.txtCurrentTime);
    myScrollView = (ScrollView) findViewById(R.id.myscroller);
    btnWhatTime = (Button) findViewById(R.id.btnWhatTime);
    btnWhatTime.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            txtCurrentTime.setText(Calendar.getInstance().getTime().toString() );
        }
    });

    mainGlobalVar.set(1);  //this value will be seen and changed by back worker

}// onCreate

public void onStart() {
    super.onStart();
    // this code creates anonymous background worker where some slow job is done
    Thread background = new Thread(new Runnable() {                                3
        public void run() {
            try {
                // experiment using/not-using isRunning flag
                for (int i = 0; i < MAX_SEC && isRunning.get(); i++) {
                    // try a Toast method here (it will not work!)
                    // fake busy busy work here
                    Thread.sleep(1000); // one second at a time
                    // this is a locally generated value between 0-100
                    Random rnd = new Random();
                    int localData = (int) rnd.nextInt(101);
```

```
                // we can see and change class variables
                String data = mainGlobalVar + " -Random Data- " + localData;
                mainGlobalVar.getAndIncrement();
                //next statement fails, note that UI is out of reach!
                //txtCurrentTime.setText("mainGlobalVar " + mainGlobalVar);

                // request a message token and put some data in it (even a Bundle)
                Message msg = handler.obtainMessage(1, (String) data);

                // if thread is still alive send the message to UI's Handler
                if (isRunning.get()) {
                   handler.sendMessage(msg);
                   Log.e("Back-Worker", data);
                }
             }
          } catch (Throwable t) {
             // just end the background thread
             isRunning.set(false);

          }
       }
    });// Tread

    isRunning.set(true);
    background.start();

  }// onStart

  public void onStop() {
     // this activity is about to die, (just in case) make sure back thread is safely stopped
     super.onStop();
     isRunning.set(false);                                                    4
  }// onStop
}// class
```

Comments

1. The `handler` object adds a safe-thread message queue to the main activity. The method `handleMessage` executes each time that a `Message` object becomes available from the queue. In such a case, it extracts from the incoming `obj` object the string message sent by the background worker. The message's text is displayed on the UI, and the linear progress bar is advanced. When `barHorizontal` reaches its completion, the linear and rotating ProgressBars (`barHorizontal` and `barCircular`) are removed from the UI. The control variable `isRunning` is set to `false`, so the background thread's loop can eventually terminate.
2. The linear progress bar `barHorizontal` is initialized, and its maximum is defined. The statement `barHorizontal.incrementProgressBy(1)` advances its value each time that a message is processed.
3. The `onStart()` method is used to start an anonymous background thread. The thread is controlled by a discrete for-loop; however, it may terminate its execution as soon as it finds that the control variable `isRunning` has been changed to `false`. While working, the thread sleeps 1000 milliseconds, then wakes up and issues the command `msg = handler.obtainMessage(1, (String) data)` requesting a

free token from the message queue. The token's "`What`" compartment is set to 1, and its object portion receives a `(String) data` message. If the control variable `isRunning` holds `true`, then the message `msg` is enqueued (`handler.sendMessage(msg)`). Observe that the shared variable `mainGlobalVar` -defined in the main thread's space as `volatile AtomicInteger`- is accessed and modified by the background worker through the safe expression `mainGlobalVar.getAndIncrement()`.

4. The `onDestroy()` method is called when MainActivity finishes. The control variable `isRunning` is changed to `false` to make sure that the background worker will discover the change and terminate as well. This precaution is important, note that a thread may continue to work after the end of its creator.

Example 9.3 Android's Handler Class – Posting

This example is functionally identical to Example 9.2. The difference between them stems from the way in which we update the app's UI. Here we use the Handler's posting strategy instead of message handling. As before the app consists of a main UI thread and a background task where some slow work is done.

The app's UI shows a button. When pressing; it immediately displays the current date on the screen (as evidence of prompt responsiveness.) The background thread periodically inserts post signals in the Handlers queue. A runnable is awoken to update UI elements each time that a token is removed from the main's queue.

Assembling App 9.3

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-03-HandlerPosting`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Use the same `res/layout/activity_main.xml` layout file of Example 9.2.

Step 3. Enter your app's code. Use the following code to establish the app's `MainActivity` class. For simplicity, the main thread, Handler, and Runnable are all defined in the `MainActivity`.

Example 9.3 MainActivity.java class (A09-03-HandlerPosting)

```
// using Handler post(...) method to execute foreground runnables
public class MainActivity extends AppCompatActivity {
   ProgressBar barHorizontal; // UI components
   ProgressBar barCircular;
   TextView txtReturned;
   TextView txtWorkProgress;
   TextView txtCurrentTime;
   Button btnWhatTime;
   Context context;
   int progressPosition = 0;
   // this main thread variable can be read-written by background threads
   volatile AtomicInteger mainGlobalVar = new AtomicInteger(0);
   // global flag to turn off threads when activity is done!
   volatile AtomicBoolean isRunning = new AtomicBoolean(true);
   int progressStep = 5;        // lenght of a simulation step
   final int MAX_PROGRESS = 100;  // length of full simulation cycle
```

```
Handler myHandler = new Handler();  // minimal Handler - no need to expand code.          1

@Override
public void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main2);
   context = getApplicationContext();
   // UI plumbing
   txtReturned = (TextView) findViewById(R.id.txtReturnedValues);
   txtWorkProgress = (TextView) findViewById(R.id.txtWorkProgress);
   barHorizontal = (ProgressBar) findViewById(R.id.progress1);
   barCircular = (ProgressBar) findViewById(R.id.progress2);
   txtCurrentTime = (TextView)findViewById(R.id.txtCurrentTime);
   btnWhatTime = (Button) findViewById(R.id.btnWhatTime);
   btnWhatTime.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View view) {
         // just to prove that we are responsive
         txtCurrentTime.setText(Calendar.getInstance().getTime().toString() );
      }
   });


}// onCreate

@Override
protected void onStart() {                                                              2
   super.onStart();
   // prepare UI components
   txtCurrentTime.setText("");
   txtReturned.setText("Posted by thread(runnable)...");

   // reset and show progress bars
   progressPosition = 0;
   barHorizontal.setMax(MAX_PROGRESS);
   barHorizontal.setProgress(0);
   barHorizontal.setVisibility(View.VISIBLE);
   barCircular.setVisibility(View.VISIBLE);

   // create background thread were the busy work will be done
   Thread myBackgroundThread = new Thread(backgroundTask, "backAlias1");
   myBackgroundThread.start();
}

// FOREGROUND
// this foreground Runnable works on behave of the background thread,
// its mission is to update the main UI which is unreachable to back workers.
// It is woken up every time a signal posted in the Handler's queue becomes available.
private Runnable foregroundRunnable = new Runnable() {                                  3
   @Override
   public void run() {
      try {
```

```
            txtWorkProgress.setText("\nPct progress: " + progressPosition + "/" + MAX_PROGRESS);
            txtReturned.append("\n mainGlobalVar: " + mainGlobalVar.get());
            // advance ProgressBar
            barHorizontal.incrementProgressBy(progressStep);
            progressPosition += progressStep;
            // are we done yet?
            if (progressPosition >= barHorizontal.getMax()) {
               txtReturned.append("\nAll Done!\n");
               txtWorkProgress.setText("All done! - Slow background work is OVER!");
               barHorizontal.setVisibility(View.INVISIBLE);
               barCircular.setVisibility(View.INVISIBLE);
            }
         } catch (Exception e) {
            Log.e("<<foregroundTask>>", e.getMessage());
         }
      }
   }; // foregroundTask

   private Runnable backgroundTask = new Runnable() {                                    4
      // BACKGROUND WORKER. This is the runnable that executes the slow work
      @Override
      public void run() {
         try {
            for (int n = 0; (n < MAX_PROGRESS/progressStep) && (isRunning.get()); n++) {
               Thread.sleep(1000);              // simulates 1 sec. of busy activity
               mainGlobalVar.getAndIncrement();  // change global protected variable here...
               // TRY. Next two UI operations should NOT work - UI is out of reach!
               // Toast.makeText(getApplication(), "Hi ", 1).show();
               // txtCurrentTime.setText("Hi ");

               // wake up foregroundRunnable delegate to speak-work for me
               myHandler.post(foregroundRunnable);

            }
         } catch (InterruptedException e) {
            Log.e("<<foregroundTask>>", e.getMessage());
         }

      }// run
   };// backgroundTask

   @Override
   protected void onStop() {
      //main activity is about to die, make sure background task is also stopped          5
      super.onStop();
      isRunning.set(false);  //try on-off -- see Log
   }

}// ThreadsPosting
```

Comments

1. The `Handler` declaration adds a blocking queue to the main thread. Observe that no one of its methods is overridden in this example. The object named `myHandler` will pass a posted token to the designated runnable when one becomes available.
2. We have chosen the `onStart()` method to reset the progress bars, text boxes, and start the execution of the slow background thread.
3. The `foregroundRunnable` object will periodically work on behalf of the background task. It is responsible for advancing the linear progress bar and updating two TextViews. When `foregroundRunnable` detects completion of the job, it removes the two progress bars from the UI.
4. The background worker simulates busy activity by sleeping blocks of 1000 milliseconds. After each block, it attaches a token to the Handler's queue through the statement `myHandler.post(foregroundRunnable)`. This could be interpreted as a signal saying "*foregroundRunnable wake up and do your job*". The runnable object has access to the UI elements and can successfully carry on the assigned task of refreshing the app's view.
5. The onDestroy() method is called to change the loop-control variable `isRunning` and set it false. This assignment, in turn, is used to terminate the background thread. Note that a thread may survive the activity that spawned it.

The AsyncTask Class

The **AsyncTask** class is a self-contained Android mechanism for multitasking that does not involve the use of `Threads` or `Handlers`. In addition, it provides a convenient built-in method for informing about the progress made by the busy parallel task.

It is particularly useful in situations in which the user must anyways wait for some busy operation to complete before interacting with the app's UI. For instance, consider actions such as transferring data over the Internet, or managing a SQL database. Note that slow operations cannot be kept in the main thread, as they would render the UI not responsive. To use an `AsyncTask`, you need to decouple the app's functionality into two pieces (1) UI maintenance logic, and (2) slow performing operations. The slow performing portion of logic is moved into the `AsyncTask`, which behaves as a thread-based background worker. What is unique to `AsyncTasks` is their ability to publish progress reports independently of the main thread. Usually, those reports inform the user of advances made by the busy task. Note that the policy prohibiting direct manipulation of the UI elements applies only to the `AsyncTask` method called `doInBackgroud()`. Other methods may update the UI.

The generic expression `new AsyncTask<Params,Progress,Result>()` instantiates and new object and identifies the data type of the input elements (Params), the variables used to publish progress (Progress), and the type of the final results (Result) obtained by the object. Table 9.2 summarizes the state methods that you need to override when using the `AsyncTask` class.

AsyncTask State Methods
`onPreExecute()` This is the first method to be called from the UI when the AsyncTask is executed. Commonly you setup here the reporting tool to be used to tell the user what is happening (usually a ProgressBar).
`doInBackground(Params...)` It follows `onPreExecute()`, and runs in a background thread where the busy job is done. This step can optionally call `publishProgress(Progress...)` to publish progress reports. These reports are published on the UI thread, during the execution of the `onProgressUpdate(Progress...)` step.
`onProgressUpdate(Progress...)` Invoked on the UI thread after a call to `publishProgress(Progress...)`. This method is used to inform of any advances made while the background computation is still executing.

`onPostExecute(Result)` Invoked on the UI thread after the background computation finishes. The result of the background computation is passed to this step as a parameter.

Table 9.2 AsyncsyncTask Methods

AsyncTask Template

```
public class MyBusyAsyncTaskTemplate extends AsyncTask<String, Long, Void> {

   public MyBusyAsyncTaskTemplate (){
      //optional - you may get a reference to MainActivity to manage its UI from here
   }

   protected void onPreExecute() {
      // optional - prepare reporting tool (usually a ProgressDialog box) - may update UI
   }

   protected Void doInBackground(final String... args) {
      // this is the SLOW background thread taking care of heavy tasks - cannot change UI
      ...
      publishProgress((Long) someLongValue);
      ...
   }

   @Override
   protected void onProgressUpdate(Long... value) {
      // periodic updates - it is OK to change UI
   }

   protected void onPostExecute(final Void unused) {
      // All done! - return results (if any) - may update UI from here
   }
}
```

The previous code template sketches the anatomy of a typical `AsyncTask<type1, type2, type3>` class implementing the four methods described in Table 9.1. Observe how the three generic types are linked to core methods. For instance, the arguments received in the `doInBackground()` method are `type1`, while `type2` is used in the `publish Progress()` and `onProgressUpdate()` methods. Finally, `type3` is used for the result values. In the case you need to skip one of the generic types (no arguments exchanged) then you should use the generic `Void` Java type.

Note The Java **Type Ellipsis** notation "`String ...`" (called **Varargs)** indicates an array of `String` values. It provides for a more flexible method calling. For instance, a call to the sum method bellow could be done using `sum(1,2,3)` instead of `sum(new Integer[]{1,2,3})`.

```
public  Integer sum(Integer... items) {
   int accum = 0;
   for (int i = 0; i < items.length; i++) {
      accum += items[i];
   }
   return accum;
}
```

Example 9.4 Using the AsyncTask class

This example is functionally similar to Example 9.2. However, its parallel operations are implemented using an `AsyncTask`. The app's UI shows a button ("Push me anytime") that immediately displays the current date on the screen (as evidence of prompt responsiveness.) The `AsyncTask` component displays a `ProgressBar` at the bottom of the screen. It includes a rotating progress bar, and a text message telling what step has been completed. The `ProgressBar` has been set to ignore outside touches and run all the way to its completion.

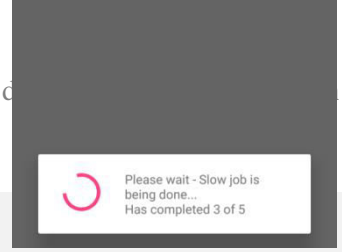
Assembling App 9.4

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-04-AsyncTask`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Modify the default `res/layout/activity_main.xml` file to be as indicated below.

Example 9.4 Layout: activity_main.xml (A09-04-AsyncTask)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:gravity="center_horizontal"
              android:orientation="vertical"
              android:padding="16dp">

    <Button
        android:id="@+id/btnSlow"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="6dp"
        android:ems="15"
        android:text="Slow work -- AsyncTask"/>

    <Button
        android:id="@+id/btnQuick"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="6dp"
        android:ems="15"
        android:text="Push me anytime!"/>

    <TextView
        android:id="@+id/txtMsg"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="6dp"
```




```
        android:background="#fcfc78"/>

</LinearLayout>
```

Step 3. Enter your app's code. Use the following code to establish the app's `MainActivity` class. For simplicity, the `AsyncTask` is defined in the `MainActivity` as an inner class.

Example 9.4 MainActivity.java (A09-04-AsyncTask)

```
public class MainActivity extends AppCompatActivity {
   Button btnSlowWork;
   Button btnQuickWork;
   TextView txtMsg;
   Long startingMillis;

   @Override
   public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      txtMsg = (TextView) findViewById(R.id.txtMsg);

      // lazy way out of multitasking - UI will run slow operations
      // StrictMode.ThreadPolicy policy = new
      //        StrictMode.ThreadPolicy.Builder().permitAll().build();
      // StrictMode.setThreadPolicy(policy);

      // slow work...for example: delete all records stored in a large database
      btnSlowWork = (Button) findViewById(R.id.btnSlow);
      this.btnSlowWork.setOnClickListener(new View.OnClickListener() {
         public void onClick(final View v) {
            //anonymous asynctask -- passing two String input arguments
            new VerySlowAsyncTask().execute("dummy1", "dummy2");                    (1)
         }
      });

      // quick response - display date-time on the UI
      btnQuickWork = (Button) findViewById(R.id.btnQuick);
      this.btnQuickWork.setOnClickListener(new View.OnClickListener() {
         public void onClick(final View v) {
            txtMsg.setText((new Date()).toString());
         }
      });

   }// onCreate


   // define the inner AsyncTask class
   private class VerySlowAsyncTask extends AsyncTask<String, Long, Void> {
      String waitMsg = "Please wait - Slow job is being done... ";
      private final ProgressDialog dialog = new ProgressDialog(MainActivity.this);

      protected void onPreExecute() {                                               (2)
         //prepare dialog box to keep user informed - we have full access to the UI thread here
         startingMillis = System.currentTimeMillis();
```

```
        txtMsg.setText("Starting date:\n" + Calendar.getInstance().getTime());
        this.dialog.setMessage(waitMsg);
        this.dialog.setCancelable(false); //don't DISMISS box with outside touches
        this.dialog.getWindow().setGravity(Gravity.BOTTOM); //show at the bottom of screen
        this.dialog.show();
    }

    // doing the busy work - UI cannot be directly modified here
    protected Void doInBackground(final String... args) {                    3
        // show on Log.e the incoming dummy arguments - for instance database name, location
        Log.e("doInBackground>>", "Total args: " + args.length );
        Log.e("doInBackground>>", "args[0] = " + args[0] );  //dummy1
        Log.e("doInBackground>>", "args[1] = " + args[1] );  //dummy2
            final long MAX_CYCLES = 5; //total simulation cycles
        try {
            // simulate here the slow activity
            for (Long counter = 1L; counter <= MAX_CYCLES; counter++) {
                Thread.sleep(5000);  //sleep for 5 seconds
                publishProgress((Long) counter, (long) MAX_CYCLES); //send progress reports
            }
        } catch (InterruptedException e) {
            Log.e("slow-job interrupted", e.getMessage());
        }
        return null;
    }

    // periodic UI updates - tell the user what is happening - you may update the UI
    @Override
    protected void onProgressUpdate(Long... progValue) {                     4
        super.onProgressUpdate(progValue);
        //receiving two arguments in args array: currentCycle and MAX_CYCLES
        long currentCycle = progValue[0];
        long totalCycles = progValue[1];
        dialog.setMessage( waitMsg
                        + "\nHas completed " + currentCycle + " of " + totalCycles);
        txtMsg.append("\nFinished step..." + currentCycle );
    }

    // we have full access to the UI here
    protected void onPostExecute(final Void unused) {                        5
        if (this.dialog.isShowing()) {
            this.dialog.dismiss();
        }
        // cleaning-up, all done
        long totalSeconds = (System.currentTimeMillis() - startingMillis) / 1000;
        txtMsg.append("\n\nElapse time: "+ totalSeconds + " sec."); //ok to change UI
        txtMsg.append("\nAll done!");
    }

  }// AsyncTask

}// MainActivity
```

Comments

1. The `MainActivity` instantiates our `AsyncTask` passing two dummy string parameters.

2. `VerySlowAsyncTask` sets a `ProgressDialog` box to keep the user aware of the advances made by the slow task. The box is defined as *not cancellable*, so touches on the UI will not dismiss it (as it would do otherwise).

3. `doInBackground` accepts the parameters supplied by the `.execute(…)` method. It fakes slow progress by sleeping various cycles of 10 seconds each. After awaking it calls the method `publishProgress()` which, in turn, asks the `onProgressUpdate()` method to refresh the `ProgressDialog` box as well as the user's UI.

4. The method `onProgressUpdate()`receives in its `Long…` array a single value coming from the background method. The arriving argument is advertised in the UI's textbox and the dialog box.

5. The `OnPostExecute()` method performs house-cleaning, in our case, it dismisses the dialog box and adds the "`Done`" message to the UI.

Android Internal Services

A commonly used strategy for decoupling parallel Android code is based on internal **Services**. An internal Service (not to be confused with web services) is a host environment or code container that can accommodate concurrent logic that (1) typically runs unattended for long periods of time, and (2) does not interact directly with the user and consequently does not present a UI. A fundamental characteristic of services is that they run in the same process space hosting the main activity. Therefore, if a standard service is going to perform slow operations (which is the norm), it needs to spawn and manage its own background threads where the slow work occurs. In this mode, the UI remains responsive to user interactions. In addition, services may continue to work even after the conclusion of the activity that started them. Therefore, it is important to guarantee a clean exit from services which are not needed any longer.

Services are also helpful in queueing situations modeling multiple clients and a busy single server. In this pattern, client's requests are serialized and chronologically inserted into a 'work queue.' When free, the server detaches from the queue the next available entry and runs without interruptions to the end of the request. For instance, a service may be used to process a sequence of petitions asking for the full downloading of data held in a distant location. Clearly, each file should be completely transferred before starting with the next request.

There are two main Service groups: `IntentServices` and `Standard Services`.

- `IntentServices` represent a new Android idiom that facilitates the quick construction of simple single-mission background workers. They run in a system-created thread that you could use for hosting slow operations.
- `Standard Services` (also called `Foreground Services`) are intended for more complex multiple-mission scenarios in which the users may ask the service to perform a variety of (typically) related operations. You need to spawn (and kill) your own background threads.

Services typically communicate with the main thread using two native Android vehicles: `Notifications` and `BroadcastRececeivers`. In the next two examples, we will learn how to create apps based on the `Service` classes, and we will explore how notifications and receivers help to support the Android's inter-process communication scheme.

Starting and stopping Services

Services can be started with the methods `startService()` and `bindService()`. However, the service is created and started only with the *first* `startService()` call. A service can issue `stopSelf()` to terminate its execution. Only one **stopService()** call is needed to end the service, no matter how many times `startService()` was called. Requests arriving at an `IntentService` are stored and process using a chronological single-server model.

Figure 9.4 illustrates the situation in which an `IntentService` receives *n* petitions. An internal work-queue stores them in arrival order until the server is ready to process the next order. The process ends when the current task is completed, and there are no more requests in the queue, or when any client issues a `stopService()` command.

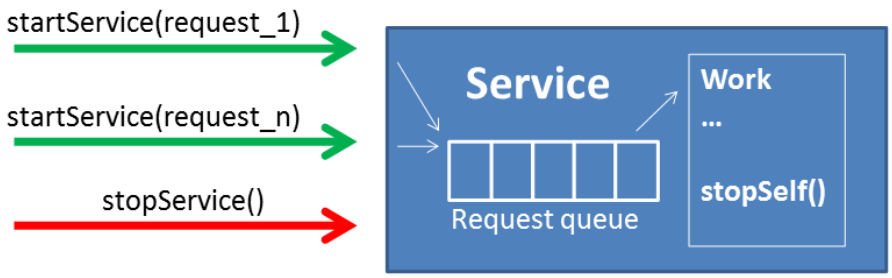


Figure 9.4 A Service enqueueing multiple requests

Service Life-Cycle

The lifecycle of a Service is much simpler than that of the `Activity` class. For simplicity, we will consider only services that are started by a client (the option to `Bind` to a service is not covered in these notes). For our discussion, assume the current process space consists of its main thread (client), a service, and other activity-type components. We will distinguish two cases based on whether they involve a plain `Service` or an `IntentService`.

Life-Cycle of a Standard Service

Calling `startService()` for the first time and passing to it an intent object creates its only possible instance. Intents carry user requests.

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

- In this case the `onCreate()` method is executed once, and when completed, it transitions to `onStartCommand()` where the incoming `Intent` is consumed.
- The service runs until it stops itself by calling `stopSelf()`. However, another component may invoke `stopService()` to end the service. Only one `stopService()` call is needed to finish the service, no matter how many times `startService()` was called.
- If a component issues a `startService()` on a service already started, it results in a direct call to its `onStartCommand()` method.
- `onDestroy()` is called when the service is no longer needed. Observe that you still have to stop any threads spawned by the service.

The Lifecycle of a started service is independent of the component that triggered it. The service can run in the background even after the termination of its creator. Figure 9.5 illustrates the main lifecycle states of a Standard Service.

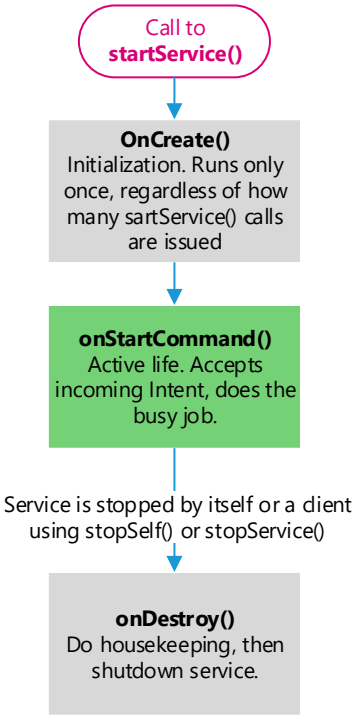


Figure 9.5 Service Lifecycle

Life-Cycle of an IntentService

The `IntentService` is a convenient class suited for the *single-server work-queue* pattern. It handles asynchronous requests wrapped in Intents. The class exposes itself as a *one-state* solution where its unique method `onHandleIntent(intent)` is the host container for your background logic. The `onHandleIntent()` method runs in a thread created by the system. Therefore it can process heavy loads without affecting the performance of the UI's thread. Android's reasoning to introduce such a class is to simplify somehow the complexity of multitasking by offering a single ready-to-use envelope to contain the app's busy parallel logic, while the system takes responsibility for creating, starting, and stopping the envelope itself.

`IntentServices` are started when a client issues a request with `startService(Intent)`. If the service is busy attending a previous request, the new job is queued in the service's FIFO queue where it waits for the completion of requests received ahead of it. The `IntentService` may short-circuit the execution of its current task, with `stopSelf()`. Then, it may continue with the next intent waiting in the internal work-queue. If no intent is found in the work-queue, the `IntentService` terminates. A call to `stopService()` will not interrupt an `IntentService` that is busy working on its `onHandleIntent()` logic.

Example 9.5 details the implementation and testing of an app based on an `IntentService` whose mission is to assume responsibility for the app's busy work while allowing the main thread to maintain an agile UI.

Broadcast Receivers

A `BroadcastReceiver` is a class that listens for intents that are broadcast at a system-wide level. For instance, when a hardware-capable device activates its Location service, its GPS coordinates are published

periodically. Any number of apps (such as mapping, news, and weather) may listen to those messages (or choose to ignore them completely.) Similarly, an app may include any number of `BroadcastReceivers`.

Global broadcast intents are implemented in Android as `IntentFilters`. Like common intents, they can carry any number of data elements. Their broadcast action is defined through a `FILTER` component. Filters are signatures used to identify the genre of the message. A `BroadcastReceiver` wanting to accept a particular kind of global intent must register with the same filter used by the broadcaster. Registering for a particular type of message could be done in a programmatic way or statically via the application's manifest. For instance, the following fragment shows a run-time filter binding operation, where `MyMainLocalReceiver` is a custom receiver, and the string "`matos.filter.MYSERVICE1`" is the filter's action tag.

```
IntentFilter intentFilterService1 = new IntentFilter("matos.filter.MYSERVICE1");
BroadcastReceiver broadcastReceiver1 = new MyMainLocalReceiver();
registerReceiver(broadcastReceiver1, intentFilterService1);
```

Alternatively, an equivalent binding of receiver and filter could be handled through the app's manifest as follows.

```
<application
    ...
    <receiver android:name=".MainActivity$MyMainLocalReceiver">
        <intent-filter>
            <action android:name = "matos.filter.MYSERVICE1" />
        </intent-filter>
    </receiver>

</application>
```

There are three Android strategies for diffusing broadcast messages.

1. **Normal broadcasts** (sent with `sendBroadcast`) are completely *asynchronous*. The message is delivered to all receivers at the same time. However, receivers run in some *unspecified* order. The message cannot be blocked as all receivers have equal opportunities.
2. **Ordered broadcasts** (sent with `sendOrderedBroadcast`) are propagated in a daisy-chained style to qualified receivers. Any receiver in the chain may opt to pass it to the next component, or abort the broadcast (`abortBroadcast`) so that it won't be propagated to the rest.
    - Ordering receivers for execution can be controlled with the **android:priority** attribute of the matching *intent-filter*;
    - Receivers with the *same priority* will be run in an *arbitrary order*.
3. **Internal Broadcasts** (sent with **LocalBroadcastManager.sendBroadcast)** releases broadcasts to receivers that are in the same app as the sender.

Example 9.5 Using the IntentService class

This example uses an `IntentService` to run busy work while maintaining a responsive UI. Figure 9.6 depicts a simulation test of this example when called in three different scenarios.

- The first case shows an `IntentService` receiving back-to-back client requests, where each job consists of eight steps. At the end of each step, the `IntentService` broadcasts the progress made. The client's

listener grabs the broadcast message and updates its UI. Please, note that in this simulation the second request arrives while the first job is only halfway completed. Therefore, the incoming intent is queued and waits for the `IntentService` to continue working for the first client. Immediately after concluding with the first request, the `IntentService` picks the waiting intent from the queue and begins working on its behalf.

- The second simulation (Figure 9.6(b)) shows the case in which work for a request is *logically* interrupted before reaching its normal termination. This interruption is accomplished by changing a Boolean control variable (*doneWithTheService*) that is observed by `onHandleIntent()`'s code. Note that `stopService()` does not affect an `IntentService` that is already executing its `onHandleIntent()` logic.
- The third and final case (Figure 9.6(c)) shows the service working on two requests arriving almost simultaneously. Before completing the execution of the first intent, the user sends a logical interruption (by changing the loop-controlling boolean variable called *doneWithTheService*), and the service stops working. However, due to the presence of a waiting intent, the service is not finished yet. It picks the delayed request from the queue runs the job to the end.

Assembling App 9.5

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-05-IntentService`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Modify the default `res/layout/activity_main.xml` file to be as indicated below.

Example 9.5 Layout: activity_main.xml (A09-05-IntentService)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:layout_width="match_parent"
              android:layout_height="match_parent"
              android:gravity="center_horizontal"
              android:orientation="vertical"
              android:padding="16dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <ProgressBar
            android:id="@+id/progressBar"
            style="?android:attr/progressBarStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"/>

        <Button
            android:id="@+id/btnClickMe"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="Click me!"/>
```

```
        <TextView
            android:id="@+id/txtDate"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_weight="2"
            android:background="#fff176"/>

    </LinearLayout>

    <View
        android:layout_width="match_parent"
        android:layout_height="1dp"
        android:layout_margin="5dp"
        android:background="@android:color/darker_gray"/>

    <Button
        android:id="@+id/btnStartService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="5dp"
        android:ems="10"
        android:text="Start Service"/>

    <Button
        android:id="@+id/btnStopService"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="5dp"
        android:ems="10"
        android:enabled="false"
        android:text="Stop Service"/>

    <ScrollView
        android:id="@+id/my_scrollview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp">

        <TextView
            android:id="@+id/txtMsg"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:background="#ffff8d"
            android:text="Broadcast Messages"

            />
    </ScrollView>

</LinearLayout>
```

Step 3. Enter your app's code. Use the following code to establish the app's `MainActivity` class. For simplicity, the `AsyncTask` is defined in the `MainActivity` as an inner class.

Example 9.5 MainActivity.java class (A09-05-IntentService)

```
public class MainActivity extends AppCompatActivity implements View.OnClickListener {
   private static final String LOG_TAG = "<<INTENTSERVICE-MAIN>>";
   Context context;
   TextView txtDate;
   TextView txtMsg;
   Button btnStartService;
   Button btnStopService;
   Button btnClickMe;
   ScrollView myScrollview;
   IntentFilter intentFilterService1;
   BroadcastReceiver broadcastReceiver1;
   int dummyArg1 = 100;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      //UI plumbing
      ActionBar actionBar = getSupportActionBar();
      actionBar.setSubtitle("Simple Service");
      context = getApplicationContext();
      txtMsg = (TextView) findViewById(R.id.txtMsg);
      txtDate = (TextView) findViewById(R.id.txtDate);
      btnStartService= (Button) findViewById(R.id.btnStartService);
      btnStopService= (Button) findViewById(R.id.btnStopService);
      myScrollview = (ScrollView) findViewById(R.id.my_scrollview);
      btnClickMe = (Button) findViewById(R.id.btnClickMe);

      btnClickMe.setOnClickListener(this);
      btnStartService.setOnClickListener(this);
      btnStopService.setOnClickListener(this);

      // Define and register a BroadcastReceiver to capture broadcast messages
      // OPTION1. In-line definition. Using an IntentFilter object
      intentFilterService1 = new IntentFilter("matos.filter.MYSERVICE1");                    1
      broadcastReceiver1 = new MyMainLocalReceiver();
      registerReceiver(broadcastReceiver1, intentFilterService1);
      // OPTION 2. Define receiver-filter in the Manifest <receiver...<intent-filter>.../>

   }//onCreate

   @Override
   protected void onDestroy() {
      super.onDestroy();
      try {
         // last chance -- logically stop the service and de-register the receiver          2
         MyService1.doneWithTheService = true;
         unregisterReceiver(broadcastReceiver1);  //it may be unregister already
         Log.e(LOG_TAG, "onDestroy() -- stopping service & unregistering receiver");

      } catch (Exception e) {
         Log.e(LOG_TAG, "onDestroy() Warning: " + e.getMessage());
```

```
    }
  }

  public class MyMainLocalReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {                    3
      // extract message from the broadcast intent
      String returnedData = intent.getStringExtra("myServiceData");
      txtMsg.append("\n" + returnedData);
      myScrollview.fullScroll(View.FOCUS_DOWN); //scroll-down, show bottom lines
    }
  }

  @Override
  public void onClick(View view) {
    if (view.getId() == btnClickMe.getId()) {
      //I'm alive, demonstrating that I'm quick and responsive (show current time & date)
      txtDate.setText(Calendar.getInstance().getTime().toString());
    }
    if (view.getId() == btnStartService.getId()){                              4
      //start service -- register Broadcast receiver
      Intent intentStartMyService1 = new Intent(this, MyService1.class);
      // service's loop-controlVar doneWithTheService is set to FALSE (work to be done!)
      MyService1.doneWithTheService = false;
      //set some data the service may need -- you may use uri-data-string and extras
      dummyArg1++;  // mark service request with this number
      intentStartMyService1.setData(Uri.parse(dummyArg1 + "-Dummy input data goes here"));
      intentStartMyService1.putExtra("arg1", dummyArg1);
      intentStartMyService1.putExtra("arg2", this.getClass().getName()); //caller's name
      startService(intentStartMyService1);

      //update UI
      txtMsg.append("\n" + dummyArg1 + "-MyService1 started -- (see LogCat)");
      btnStopService.setEnabled(true);
      Log.e(LOG_TAG, dummyArg1 + "-service started & receiver registered");
    }

    if (view.getId() == btnStopService.getId()){
      try {
        // interrupting service by assigning TRUE to its loop-controlVar doneWithTheService
        txtMsg.append("\n" + dummyArg1 + "-Stop service...");
        MyService1.doneWithTheService = true;                                  5
        Log.e(LOG_TAG, dummyArg1 + "- stop service & de-register receiver");

      } catch (Exception e) {
        Log.e(LOG_TAG, dummyArg1 + "-btnStopService ERROR " + e.getMessage());
      }
    }//onClick

  }//onCreate

}
```

Comments

1. Use `registerService(listener, filter)` to create an inline association between the listener and its filter. The `IntentFilter` string "`matos.filter.MYSERVICE1`" represents the broadcast action that will be used to trigger the receiver's listener. A sender must use this filter to identify its messages and activate the listener's instance hosted in the `MainActivity`.
2. We use `onDestroy()`to stop the service and de-register the broadcast listener. Note that if the service is running its `onHandleIntent`, then issuing `stopService` is not sufficient to cancel operations (it will be ignored). Instead, we have coded the service's method in such a way that it constantly looks for changes of the control variable *`doneWithTheService`*. When the variable changes to `true` the service eventually stops.
3. `MainLocalReceiver` is the class from which the listener in the main thread is made. When a message tagged with the id "`matos.filter.MYSERVICE1`" is released, the listener grabs a copy, extracts its contents and updates the UI accordingly.
4. Each time the user taps on the "**Start Service**" button, a new common intent is created. After the intent is loaded with data, it is used as an argument of the `startService` command, which either initiates a new instance of the `IntentService` or appends the current request to the service's work-queue.
5. Tapping the "**Stop Service**" button changes the control variable *`doneWithTheService`* to `true`. The new value will ultimately be detected in the service forcing the termination of `onHandleIntent`. The service stops when no other request is waiting for attention in the work-queue.

Step 4. Define your IntentService class. Use the following code to create your `IntentService` subclass.

Example 9.5 MyService1.java class (A09-05-IntentService)

```
public class MyService1 extends IntentService {
   // doneWithTheService is used to stop the service. Our logic terminates when it becomes TRUE.
   // Notice the variable is atomic, static (only one for the class), and public (seen outside)
   public static volatile boolean doneWithTheService = false;

   private static String LOG_TAG = "<<INTENT_SERVICE>>";

   public MyService1() {
      // A constructor is required, and must call the super IntentService(String)
      // constructor with a name for the worker thread.
      super("MyService1");
      Log.e(LOG_TAG, "Service1 -- after constructor");
   }

   @Override
   protected void onHandleIntent(Intent intent) {
      // grab input uri-data-string & extras from incoming intent
      String inputData = intent.getDataString();
      int arg1 = intent.getIntExtra("arg1", 0);
      String arg2 = intent.getStringExtra("arg2");
      Log.e(LOG_TAG, "Input data:" + inputData + " arg1:" + arg1 + " arg2:" + arg2);
```

```
      // Normally we would do some single-mission slow work here (one request at the time).
      // In our sample we just sleep for a few seconds.
      Intent myBroadcastMessageIntent = null;  //this intent will carry the broadcast message
      doneWithTheService = false;              //open the service gate to this request
      try {
         // IntentService automatically ends when this method ends. However, if triggering
         // activity ends without notifying the service, then the service continues indefinitely.
         // Note the control variable doneWithTheService may short-circuit the loop and terminate
         // the method at any time.
         for (int i = 1; (i <= 8) && (!doneWithTheService); i++) {
            Thread.sleep(500); //fake busy work
            //prepare and send broadcast message to all interested listeners
            myBroadcastMessageIntent = new Intent("matos.filter.MYSERVICE1");
            String msg = arg1 + "-some result data goes here " + i;
            myBroadcastMessageIntent.putExtra("myServiceData", msg);
            sendBroadcast(myBroadcastMessageIntent);
            Log.e(LOG_TAG, arg1 + "-Service1 -- after sleeping/working " + i);
         }
         stopSelf(); // terminate service instance
         myBroadcastMessageIntent.putExtra("myServiceData", arg1 + "-All done!");
         sendBroadcast(myBroadcastMessageIntent);
         Log.e(LOG_TAG, arg1 + "-Service1 -- stopped");

      } catch (InterruptedException e) {
         // Restore interrupt status.
         Thread.currentThread().interrupt();
         Log.e(LOG_TAG, arg1 + "-Service1 -- after Thread.currentThread().interrupt() ");

    }
   }//onHandleIntent

}//MyService1
```

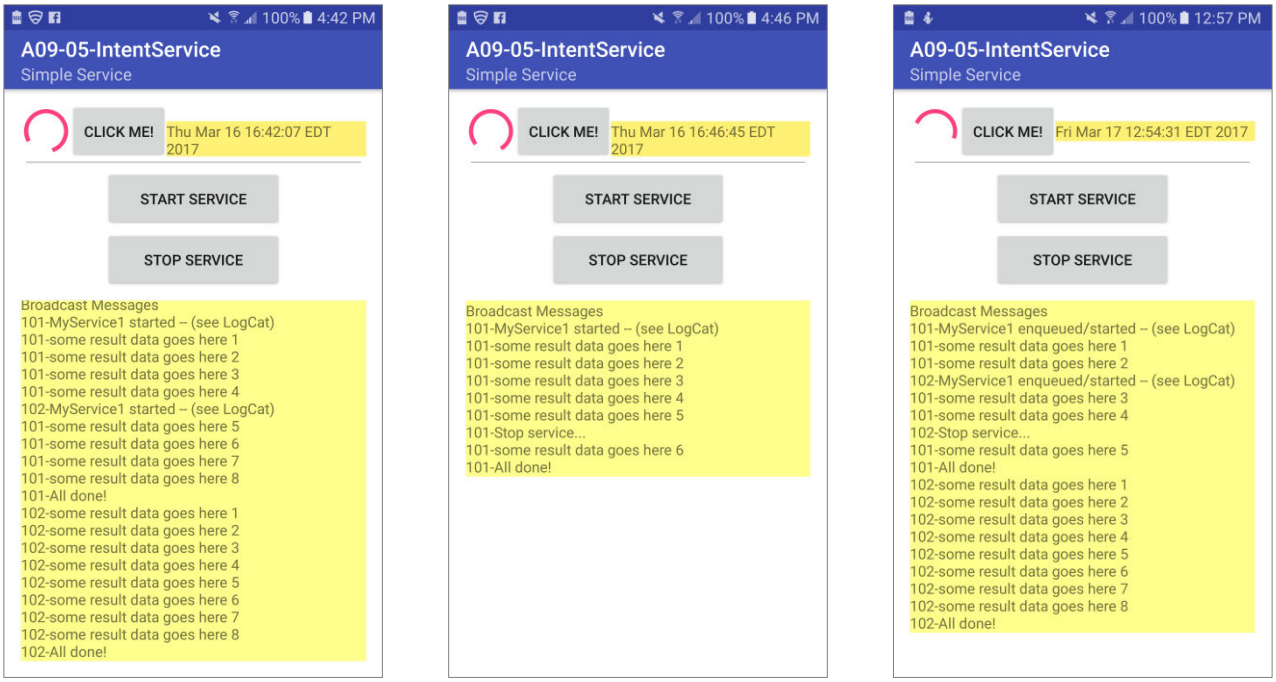
Step 5. Setup the app's Manifest. Make sure your Manifest includes a clause defining the service to be called. Observe the optional static specification for our `BroadcastReceiver`.

Example 9.5 AndroidManifest.xml (A09-05-IntentService)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest package="matos.csu.a09_05_intentservice"
          xmlns:android="http://schemas.android.com/apk/res/android">

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
```

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

    <service android:name=".MyService1"/>

    <!--The following is an option to register a BroadcastReceiver     -->
    <!--that has been created as an independent class (separate file)  -->
    <!--It is NOT needed if you do programmatic in-line registration   -->
    <receiver android:name=".MainActivity$MyMainLocalReceiver">
        <intent-filter>
            <action android:name = "matos.filter.MYSERVICEX" />
        </intent-filter>
    </receiver>

</application>

</manifest>
```

Note that each service started by the application must be registered in the `AndroidManifest` using a `<service>` tag. The `<receiver>` clause together with its `<intent-filter>` is optional. Alternatively, you may opt to define both tags programmatically.



(a)IntentService serializes two requests

(b) Stopping a service request

(c) Stopping first of two service requests

Figure 9.5 Two instances of the `IntentService` app

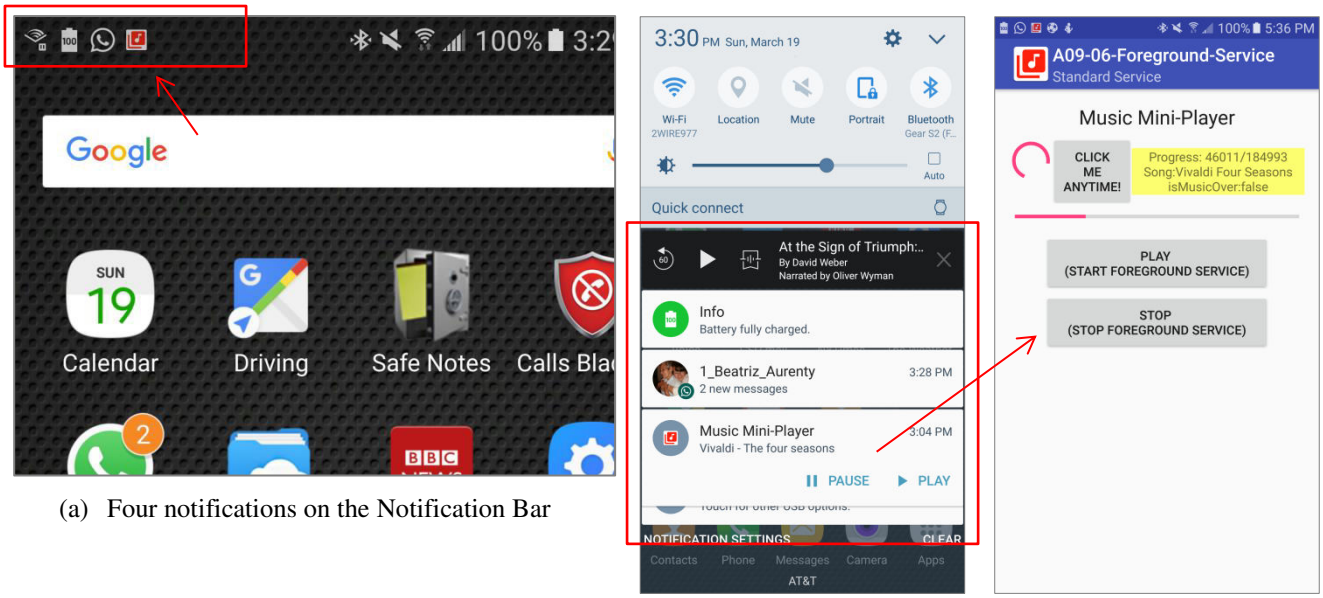
Notifications

The Android Notification scheme is an important strategy to call the user's attention using a minimalist approach to UI intrusion. A notification is a very subtle visual delegate that sits on top of the screen consuming almost no resources. It is especially suited to work with background services, which do not have a direct UI presence.

At first, a notification appears as a small icon in the *device's notification area*. In Android, the screen's top area is called the *system bar*. This bar is shared by the *status area* (right side) and the *notification area* (left side). The status area typically displays the local time, battery level, and various icons such as level of tower signal, Bluetooth, WIFI, volume, and so on. The notification bar contains a set of icons, each typically drawn as a stylized version of the app's logo.

You may ignore a notification and let it stay on the bar, or choose to see its details as soon as it appears. To explore them, you need to pull down the status bar, which displays the *notification drawer*. The drawer is a list holding a row for each icon in the status bar. Once the drawer is opened, you may begin a gesture-driven interaction with the available notifications. For instance, you may scroll the list (up/down), dismiss an entry (sideswipe), clear all entries, expand an entry (drag-down a row), and touch portions of a fully expanded notification (embedded buttons).

Figure 9.6(a) shows a device that has received four notifications. Observe they are stacked in the top-left corner of the screen. Figure 9.6(b) displays the corresponding notification drawer. Most cells in the notification list have a similar layout (icon on the left, then a top title, description line, and optional buttons). A row may also have a completely customized appearance (see the first row, showing an Audible© app notification). Figure 9.6(c) depicts the activity called after tapping the body of the Music Mini-Player notification (this app will be fully developed in Example 9.6.)



(a) Four notifications on the Notification Bar

(b) Notification Drawer

(c) Music Mini-Player

Figure 9.6 Using a notification to run the Music Mini-Player app

A notification object must include three mandatory components (1) a small icon, (2) a title, and (3) a detail-text. There are other optional parts -most notably actions buttons- that also could be attached to a notification. The following code fragment defines the *Music Mini-Player* notification shown in Figure 9.6(b)

Generating a notification object for the Music Mini-Player app

```
private void prepareNotification() {
   //this is the (main) blown-up activity to be shown by the notification
   Intent notificationIntent = new Intent(this, MainActivity.class);
   notificationIntent.setAction(MyConstants.MAIN_ACTION);
   notificationIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
                              |Intent.FLAG_ACTIVITY_CLEAR_TASK);
   //to be used in case the user touches the notification body
   PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);
   //to be used in case the user touches the notification's PAUSE button
   Intent pauseIntent = new Intent(this, MusicService.class);
   pauseIntent.setAction(MyConstants.PAUSE_ACTION);
   PendingIntent pausePendingIntent = PendingIntent.getService(this, 0, pauseIntent, 0);
   //to be used in case the user touches the notification's PLAY button
   Intent playIntent = new Intent(this, MusicService.class);
   playIntent.setAction(MyConstants.PLAY_ACTION);
   PendingIntent playPendingIntent = PendingIntent.getService(this, 0, playIntent, 0);

   //prepare the notification panel, its icon will be placed on the top system status line
   Notification notification = new NotificationCompat.Builder(this)
         .setContentTitle("Music Mini-Player")
         .setContentText("Vivaldi - The four seasons")
         .setSmallIcon(R.mipmap.ic_launcher)
         .setContentIntent(pendingIntent)
         .setOngoing(true)
         .addAction(R.drawable.ic_stat_pause, "Pause", pausePendingIntent)
         .addAction(R.drawable.ic_stat_play, "Play", playPendingIntent).build();
   startForeground(MyConstants.NOTIFICATION_ID_FOREGROUND_SERVICE, notification);

}//prepareNotification
```

For each actionable part of the notification in Figure 9.6(b) (body, "Play", "Pause" buttons) we prepare a `PendingIntent`. This class is a wrapper for intents that are going to be performed at a later time by another component using the same rights and privileges of the one who created the `PendingIntent`. For instance, assume the following events have happened

1. The *Music Mini-Player* activity has obtained permission to access the music files.
2. The user hits the "Play" button. The app starts a service to play Vivaldi's music, and the method above (`prepareNotification`) is executed. This results on a new notification icon placed on the notification bar (Figure 9.6(a))
3. The user hits the Back-Button terminating the activity; however, the service continues to play music.
4. The user drags down the system bar, exposing the notification drawer (Figure 9.6(b)). Then she taps on the notification body. This event fires the `PendingIntent` associated with touching the notification's body resulting in a fresh execution of the app's MainActivity (Figure 9.6(c)).
5. From the MainActivity the user hits the "Stop" button. This action ends the service's execution. The notification icon is removed from the system bar. The user terminates the application.

Example 9.6 Using the Standard Service class

In this example, we implement the bare bones Music Mini-Player app discussed in the previous section (Notifications).

The app's implementation is based on a programming style called **Foreground Service Pattern**. Customarily, this model includes a subclass of Service (i.e. ForegroundService) and one or more Notifications. In this pattern, the service runs indefinitely in the background without any UI to control it. However, it could be brought back to the user's attention using a notification object. Figure 9.6 shows various stages of the app's usage, Figure 9.7 provides a graphical representation of the classes and messages used by the Music Mini-Player app. Our implementation includes the following components

- The main thread handling the app's UI,
- A background service responsible for the actual playing of the selected music file,
- A notification object, which adds a tiny element of presence on the device's UI. This un-obstructive icon allocated in the system bar could be used to recall the player's main activity or directly operate on the service without recalling the main thread,
- A monitoring class that periodically checks the progress made by the background service.

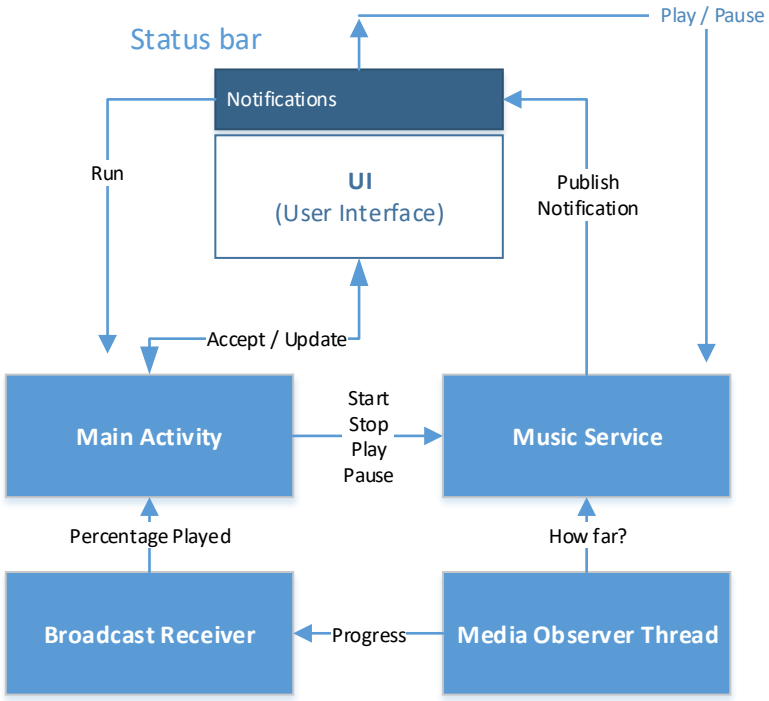


Figure 9.7 Classes and messages used by the Music Mini-Player app.

Note that we have chosen for this case a `Standard/Foreground Service` instead of a simple `IntentService`. The reason for this decision is that standard services can be interrupted. Therefore, they are more responsible and can accept and process a variety of request as soon as they arrive. For instance, consider that you give the order to play a file. Assume that shortly after you begin listening to the music you change your mind and decide to cancel it. Obviously, you would like the service to process the "Stop" request as soon as it is received. On the other hand, note that `IntentServices` cannot be interrupted. Therefore, if one chooses this kind of class, a piece of music once started, will be played until it ends (which may not be exactly an ideal behavior).

For the sake of simplicity, our Music Mini-Player is linked to a single musical selection. The service will process three different actions *Play*, *Pause*, and *Stop*. The main activity does not need to be running for the

service to work. The service is responsible for publishing a notification object. Once music playing has started, the UI thread could be terminated. To bring the UI activity back, the user needs to pull down the notification drawer and select one of the actions offered by the app's notification. Touching the notification's body re-executes the main activity. The other action buttons are "Pause" and "Play".

Assembling App 9.6

Step 1. Create the app's skeleton. Use the Android-Studio app wizard to create a new Android Project, let's call it `A09-06-ForegroundService`. On the wizard's screen labeled "Target Android Devices" make sure you choose (under the tag 'Phone and tablet > Minimum SDK') an *earlier* OS version such as SDK4.x (KitKat) or SDK5.x (Lollipop). On the next screen, choose the "`Empty Activity`" template.

Step 2. Prepare the main UI. Modify the default `res/layout/activity_main.xml` file to be as indicated below. The UI includes a circular and a linear progress bar, as well as a button "Click me anytime" that invites you to test the UI's quickness. The buttons "Start" and "Stop" will correspondingly initiate and terminate the app's supporting service.

Example 9.6 Layout: activity_main.xml (A09-06-ForegroundService)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:gravity="center_horizontal"
    android:orientation="vertical"
    android:padding="16dp">

    <TextView
        android:id="@+id/textView1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Music Mini-Player"
        android:textAppearance="?android:attr/textAppearanceLarge"/>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal">

        <ProgressBar
            android:id="@+id/progressBar"
            style="?android:attr/progressBarStyle"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"/>

        <Button
            android:id="@+id/btnWhatTime"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
```

```
            android:text="Click\nme\nanytime!"/>

        <TextView
            android:id="@+id/txtMsg"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="10dp"
            android:background="#fdfd70"
            android:gravity="center_horizontal"
            android:text="date is..."/>

    </LinearLayout>

    <ProgressBar
        android:id="@+id/progressBar2"
        style="?android:attr/progressBarStyleHorizontal"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_margin="6dp"/>

    <Button
        android:id="@+id/btnStart"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:ems="15"
        android:text="PLAY \n(Start Service)"/>

    <Button
        android:id="@+id/btnStop"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:ems="15"
        android:text="STOP \n(Stop Service)"/>

</LinearLayout>
```

Step 3. Define constants. Create a utility class (`MyConstants`) to hold all the constant values to be used by the application.

Example 9.6 MyConstants.java (A09-06-ForegroundService)

```
public class MyConstants {
   //global constants (actions, broadcast filter, notification number)
   public static String PACKAGE_NAME = "matos.csu.a09_06_foreground_service";
   public static String MY_MUSIC_FILTER = PACKAGE_NAME + ".filter.action.VIVALDI_STATION";
   public static String START_FOREGROUND_ACTION = PACKAGE_NAME + ".action.start_foreground";
   public static String STOP_FOREGROUND_ACTION = PACKAGE_NAME + ".action.stop_foreground";
   public static String MAIN_ACTION = PACKAGE_NAME + ".action.main";
   public static String PLAY_ACTION = PACKAGE_NAME + ".action.play";
   public static String PAUSE_ACTION = PACKAGE_NAME + ".action.pause";
   public static int NOTIFICATION_ID_FOREGROUND_SERVICE = 1234;
}
```

Step 4. Enter your app's code. Use the following fragment to define the app's `MainActivity` class. The activity is responsible for UI operations, and the starting/stopping of its associated `MusicService` class. Two progress bars are displayed, one of them tracks the progress made in playing the selected piece of music.

Example 9.6 MainActivity.java (A09-06-ForegroundService)

```
public class MainActivity extends AppCompatActivity implements OnClickListener {
   private static final String LOG_TAG = "<<MAIN_ACTIVITY>>";
   Button btnStartButton;
   Button btnStopButton;
   Button btnWhatTime;
   TextView txtMsg;
   ProgressBar progressBarLinear;
   IntentFilter musicIntentFilter;
   BroadcastReceiver broadcastReceiver;
   Intent startIntent;
   Intent stopIntent;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      //decorate ActionBar
      ActionBar actionBar = getSupportActionBar();
      actionBar.setSubtitle("Vivaldi's Four Seasons");
      actionBar.setIcon(R.mipmap.ic_launcher);
      actionBar.setDisplayShowHomeEnabled(true);
      actionBar.show();
      //UI Plumbing
      txtMsg = (TextView) findViewById(R.id.txtMsg);
      progressBarLinear = (ProgressBar) findViewById(R.id.progressBar2);
      btnStartButton = (Button) findViewById(R.id.btnStart);
      btnStopButton = (Button) findViewById(R.id.btnStop);
      btnWhatTime = (Button) findViewById(R.id.btnWhatTime);
      btnStartButton.setOnClickListener(this);
      btnStopButton.setOnClickListener(this);
      btnWhatTime.setOnClickListener(this);

      // (In-line option) Define filter for local listener & register receiver
      musicIntentFilter = new IntentFilter(MyConstants.MY_MUSIC_FILTER);                1
      broadcastReceiver = new MyMainLocalReceiver();
      registerReceiver(broadcastReceiver, musicIntentFilter);

   }

   @Override
   public void onClick(View v) {
      switch (v.getId()) {
         case R.id.btnStart:
            // START service (play music)
            startIntent = new Intent(MainActivity.this, MusicService.class);          2
            startIntent.setAction(MyConstants.START_FOREGROUND_ACTION);
            startService(startIntent);
```

```
            break;
         case R.id.btnStop:
            // STOP service
            stopIntent = new Intent(MainActivity.this, MusicService.class);                    3
            stopIntent.setAction(MyConstants.STOP_FOREGROUND_ACTION);
            startService(stopIntent);
            break;
         case R.id.btnWhatTime:
            // (Responsive UI) Quickly update the UI with current date-time value
            txtMsg.setText(Calendar.getInstance().getTime().toString());
            break;
         default:
            break;
      }

   }//onClick

   public class MyMainLocalReceiver extends BroadcastReceiver {                               4
      // receive messages broadcast with the "matos.action.GO_MUSIC_SERVICE" filter
      @Override
      public void onReceive(Context context, Intent intent) {
         //extract and show returned data, update progress bar
         String returnedData = intent.getStringExtra("progress");
         txtMsg.setText(returnedData);
         int duration = intent.getIntExtra("duration",1);
         int current = intent.getIntExtra("currentPosition", 1);
         Log.e(LOG_TAG, "duration " + duration);
         Log.e(LOG_TAG, "current " + current);
         progressBarLinear.setMax(duration);
         progressBarLinear.setProgress(current);
         txtMsg.append("\nSong:" + intent.getStringExtra("songName"));
         txtMsg.append("\nisMusicOver:" + intent.getBooleanExtra("isMusicOver",false));
      }
   }

   @Override
   protected void onDestroy() {
      super.onDestroy();
      try {
         // stopping the service and un-registering the receiver                               5
         MusicService.doneWithTheService = true;
         unregisterReceiver(broadcastReceiver);
         Log.e(LOG_TAG, " stopping service & un-registering receiver");

      } catch (Exception e) {
         Log.e(LOG_TAG, " ERROR " + e.getMessage());
      }

   }//onDestroy

}//MainActivity
```

Comments

1. A receiver derived from the custom class `MyBroadcastReceiver` is registered to listen to messages matching the filter `MyConstants.MUSIC_FILTER`.
2. To initiate our `MusicService` worker, we prepare an intent, set its action to the value `MyConstants.START_FOREGROUND_ACTION`, and issued `startService(intent)`.
3. Stopping a started service requires an intent carrying the action `MyConstants.STOP_FOREGROUND_ACTION`. The new intent is delivered by `startService(intent)` method. If the service is not playing the order is ignored. Otherwise, the music player instance is destroyed. Pressing "Stop" button removes the app's notification (if one has been published).
4. The `onReceive()` method in `MyBroadcastReceicer` class uses the incoming intent argument to extract data sent by the background service. The UI is updated with the returned values, and the progress bar is advanced to signal progress made in playing the music file.
5. Pressing the Back-Button stops the application and deregisters the receiver.

Step 5. Define the foreground service. Use the following fragment to define `MyMusicService` class. This standard service is responsible for accepting intents requesting four different type of actions (1) Start service, (2) Stop service and its music playing, (3) Play music, (4) Pause music playing. The service will assemble and publish a notification as soon as it begins to play a selection.

Example 9.6 MusicService.java (A09-06-ForegroundService)

```
public class MusicService extends Service implements MediaPlayer.OnPreparedListener,
                                                     MediaPlayer.OnCompletionListener {

   //var mustStopService could be used by calling activities to stop service! (not used here)
   public static volatile boolean doneWithTheService = false;

   private static final String LOG_TAG = "<<MusicService>>";
   private static boolean isStarted = false;
   private static boolean isMusicOver = false;
   private MediaPlayer mediaplayer = null;

   @Override
   public void onCreate() {                                                          1
      super.onCreate();
      Log.e(LOG_TAG, "onCreate -- first time...");
      doneWithTheService = false;
      busyWork();

   }

   private void busyWork() {                                                         2
      try {
         // WARNING ---------------------------------------------------------------------------
         // Normally this section should be a Thread separated from the UI where the slow work is
         // taken care of. Syntactically you would typically write something like this:
         //    threadHostingSlowJob = new Thread() {
         //       @Override
         //       public void run() {
         //          super.run();
         //          ... your logic for the slow job goes here ...
```

```
        //       }
        //    };
        // However, MediaPlayer offers a special case, as it runs in its own system-thread.
        // It is NOT needed from you to encapsulate its work in a custom-made Thread as the
        // above (you may still do it but at the cost of redundancy!)
        // -------------------------------------------------------------------------------
        // setup MediaPlayer to play a piece of music stored in res/raw (it could be anywhere)
        mediaplayer = MediaPlayer.create(getApplicationContext(), R.raw.vivaldi_track_01);
        mediaplayer.setOnPreparedListener(this);  //to be called when mediaplayer is ready
        mediaplayer.setOnCompletionListener(this); //fired when music playing is over
        mediaplayer.prepareAsync();  //prepare the player without blocking main UI

    } catch (Exception e) {
        Log.e(LOG_TAG, "mediaplayer.prepareAsync() failed. ERROR:" + e.getMessage());
    }

}//busyWork

@Override
public void onPrepared(final MediaPlayer mediaPlayer) {                    3
    //this Listener is called when MediaPlayer has finished preparation and is ready
    mediaPlayer.start();
    isStarted = true;
    isMusicOver = false;
}

@Override
public void onCompletion(MediaPlayer mediaPlayer) {                        4
    //the playing of current song is over
    Log.e(LOG_TAG, "onCompletion <<<<<<<< MUSIC IS OVER ");
    isMusicOver = true;
    broadcastProgress();
}

@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    //switchboard -- branch on intent.action
    Log.e(LOG_TAG, "onStartCommand >>> " + intent.getAction());

    if (intent.getAction().equals(MyConstants.START_FOREGROUND_ACTION)) {  5
        Log.e(LOG_TAG, "Received Start Foreground Intent " + " isStarted " + isStarted);
        if (!isStarted) {
            prepareNotification();
            isStarted = true;
            // The following thread will watch the mediaplayer, broadcasting periodic progress
            new Thread(new MediaObserver()).start();
        } else {
            mediaplayer.start();

        }

    } else if (intent.getAction().equals(MyConstants.PAUSE_ACTION)) {
        Log.e(LOG_TAG, "Clicked PAUSE");
```

```
        if (mediaplayer != null) {
            mediaplayer.pause();

        }

    } else if (intent.getAction().equals(MyConstants.PLAY_ACTION)) {
        Log.e(LOG_TAG, "Clicked PLAY");
        if (mediaplayer != null) {
            mediaplayer.start();

        }


    } else if (intent.getAction().equals(MyConstants.STOP_FOREGROUND_ACTION)) {
        Log.e(LOG_TAG, "Received STOP Foreground Intent");
        if (mediaplayer != null) {
            isStarted = false;
            mediaplayer.stop();
            mediaplayer.release();
            stopForeground(true);  //mediaplayer is killable
            stopSelf();

        }
    }
    return START_STICKY;
}//onStartCommand


@Override
public void onDestroy() {
    super.onDestroy();
    Log.e(LOG_TAG, "In onDestroy");
    doneWithTheService = true;  //not used here - (just in case)
    isStarted = false;
    if (mediaplayer != null) mediaplayer.release();

}

@Override
public IBinder onBind(Intent intent) {
    // Needed by the Interface -- Used only in when binding to a started service
    Log.e(LOG_TAG, "In onBind");
    return null;
}

private void prepareNotification() {                                    6
    // prepare and launch notification
    Log.e(LOG_TAG, "In prepareNotification");
    //this is the blown-up activity to be shown by the notification
    Intent notificationIntent = new Intent(this, MainActivity.class);
    notificationIntent.setAction(MyConstants.MAIN_ACTION);
    notificationIntent.setFlags(Intent.FLAG_ACTIVITY_NEW_TASK
          |                     Intent.FLAG_ACTIVITY_CLEAR_TASK);
```

```
   //to be used in case the user touches the notification body (MAIN_ACTION)
   PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

     //to be used in case the user touches the notification's pause button (PAUSE_ACTION)
   Intent pauseIntent = new Intent(this, MusicService.class);
   pauseIntent.setAction(MyConstants.PAUSE_ACTION);
   PendingIntent pausePendingIntent = PendingIntent.getService(this, 0, pauseIntent, 0);

   //to be used in case the user touches the notification's play button (PLAY_ACTION)
   Intent playIntent = new Intent(this, MusicService.class);
   playIntent.setAction(MyConstants.PLAY_ACTION);
   PendingIntent playPendingIntent = PendingIntent.getService(this, 0, playIntent, 0);

   //prepare the notification panel, its icon will be placed on the top system status line
   Notification notification = new NotificationCompat.Builder(this)
         .setContentTitle("Music Mini-Player")
         .setContentText("Vivaldi - The four seasons")
         .setSmallIcon(R.mipmap.ic_launcher)
         .setContentIntent(pendingIntent)
         .setOngoing(true)
         .addAction(R.drawable.ic_stat_pause, "Pause", pausePendingIntent)
         .addAction(R.drawable.ic_stat_play, "Play", playPendingIntent).build();

   startForeground(MyConstants.NOTIFICATION_ID_FOREGROUND_SERVICE, notification);
/*
startForeground: Make this service run in the foreground,
supplying the ongoing notification to be shown to the user while in this state.
By default services are background, meaning that if the system needs to kill them
to reclaim more memory (such as to display a large page in a web browser),
they can be killed without too much harm. You can set this flag if killing your service
would be disruptive to the user, such as if your service is performing
background music playback, so the user would notice if their music stopped playing.
 */
}//prepareNotification


private class MediaObserver implements Runnable {                                    7
   private AtomicBoolean stop = new AtomicBoolean(false);

   @Override
   public void run() {
      try {
         //sleep for 2 seconds, then wakeup, look at the MediaPlayer and report its progress
         while (!isMusicOver) {
            Thread.sleep(2000);
            broadcastProgress();

         }
      } catch (InterruptedException e) {
         Log.e(LOG_TAG, "MediaObserver ERROR: " + e.getMessage());
      }
   }
```

```
   }


   private void broadcastProgress() {                                    8
      //broadcast MediaPlayer progress to any listener tune-up to MY_MUSIC_FILTER
      try {
         int currentPosition = mediaplayer.getCurrentPosition();
         int duration = mediaplayer.getDuration();
         Intent myFilteredResponse = new Intent(MyConstants.MY_MUSIC_FILTER);
         String msg = "Progress: " + currentPosition + "/" + duration;
         myFilteredResponse.putExtra("progress", msg);
         myFilteredResponse.putExtra("currentPosition", currentPosition);
         myFilteredResponse.putExtra("duration", duration);
         myFilteredResponse.putExtra("songName", "Vivaldi Four Seasons");
         myFilteredResponse.putExtra("isMusicOver", isMusicOver);
         sendBroadcast(myFilteredResponse);
         Log.e(LOG_TAG, ">>>>>>>> Percentage: " + (100 * currentPosition / duration));
      } catch (Exception e) {
         Log.e(LOG_TAG, "showProgress " + e.getMessage());
      }

   }//showProgress


}//MusicService
```

Comments

1. The `onCreate()` method calls the `busywork()` method where all the heavy tasks of this service are encapsulated. The Boolean variable *`doneWithTheService`* is initialized (`False`). Although it is not used in this example, it is a common programming technique to introduce such kind of control variables in the conditional-expression of loop-patterned threads. When the control variable changes to true, the loop terminates, and the thread gracefully ends.
2. Our `busywork()`method represents the slow background work for which the service is responsible. Typically you would be spawning a new thread to take care of the job. The specifics about what to do would appear in the thread's `run()` method. However, this example presents a very particular case. Observe that the task of sounding the music files is given to the `MediaPlayer` class. `MediaPlayer` is a native Android artifact that simplifies manipulation of multimedia material (music, audio files, and video.) When a `MediaPlayer` is instantiated, it is automatically placed in a system created thread. As a result, the actions of the new instance does not make its creator appear as a non-responsive application. In our example, we create the `mediaPlayer` object and assign to it a particular raw file (*`vivaldi_track_01`*). Then we attach to `mediaPlayer` two asynchronous listeners to tell when the `mediaPlayer` has completed its preparation and is ready to play (`setOnPreparedListener`). The second listener becomes active when the playing action is over (`setOnCompletionListener`). Finally, we tell the listener to get ready to work in an asynchronous manner; this means that the UI thread will not be affected by a potential slow preparation of the `mediaPlayer`.
3. `onPrepared()` is called when the `mediaPlayer` is ready to work. We apply the `.play()` method and setup the control Boolean variables `isPlaying` and `isMusicOver` to `true` and `false` respectively.

4. `onCompletion()` is called when the playing of the given media file has reached its end. At this point, we change the control variable is `MusicOver` to true and invoke the method `broadcastProgress()` to tell about the completion of the task.
5. `onStartCommand()` is the service's headquarter. Once a service instance has been created, all incoming intents are accepted and processed by this state. In our problem, an incoming intent represents a user request. There are four possible actions to be attended by our bare-bone music service,
   `START_FOREGROUND_ACTION` prior to playing music for the first time, we assemble and publish a notification, and the control variable `isPlaying` is set to `true`. A thread hosting a custom `MediaObserver` object is launched to periodically monitor the progress made by the media player.
   `PAUSE_ACTION` if the `mediaPlayer` object exists, we execute the statement `mediaPlayer.pause()`. `mediaPlayer` creates a bookmark to remember the last position played in case it needs to continue.
   `PLAY_ACTION` if the `mediaPlayer` object exists, we execute the statement `mediaPlayer.start()`. Playing starts from the last bookmarked position.
   `STOP_ACTION` this terminates the service's session. If the `mediaPlayer` object exists, music playing stops and any acquired resources (such as speakers) are released. The statement `stopForeground(true)` removes the notification icon from the system bar, and marks the service as a 'killable' component (in case there is need to re-possess its memory space). Finally the service stops itself (`stopSelf()`).
6. `prepareNotification()` assembles a notification that includes an icon similar to the application's logo, a title, a line telling the song's name, and two buttons (Pause and Play). Details about how the notification works are given in the preceding section (Notifications). When the user interacts with the Music Mini-Player notification there are three possible actions to process
   (1) Tapping the body of the notification launches an intent that executes the app's main activity one more time (however, it will find the service is running, and the service's `onCreate` will be bypassed),
   (2) Clicking on the Pause button momentarily pauses music playing,
   (3) Pressing the Play button re-starts the playing of the music file.
7. `MediaObserver` is a custom runnable object to be executed inside a user's spawned thread. Its goal is to monitor the progress made by the music player and update a progress bar on the application's UI. It is implemented as a controlled iteration (variable `isMusicOver` must be `false`). Each cycle consists of two steps. First, it sleeps for 2 seconds then it asks the media player for the duration of the song and its current position. It passes this information to a `BroadcastReceiver` already registered in the `MainActivity` which will modify the user's UI.
8. `broadcastProgress` is the method responsible for assembling and releasing an intent to any `BroadcastReceivers` monitoring `MY_MUSIC` filter.

Note **Looking the other way – Allowing slow work in the main thread**

Your application's main thread should remain responsive at all times, failure to do so generates ANR messages (**A**pplication **N**ot **R**esponding). However, you may temporarily bypass your duty to write well-crafted code, through the following escape solution.

```
StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder()
                                                .permitAll().build();
```

```
StrictMode.setThreadPolicy(policy);
```

This subterfuge will let you perform slow operations that compromise the UI's agility inside the main thread. Be aware that it is a very poor programming practice. Instead, you should use threads, runnables, handlers, services, or AsyncTasks.