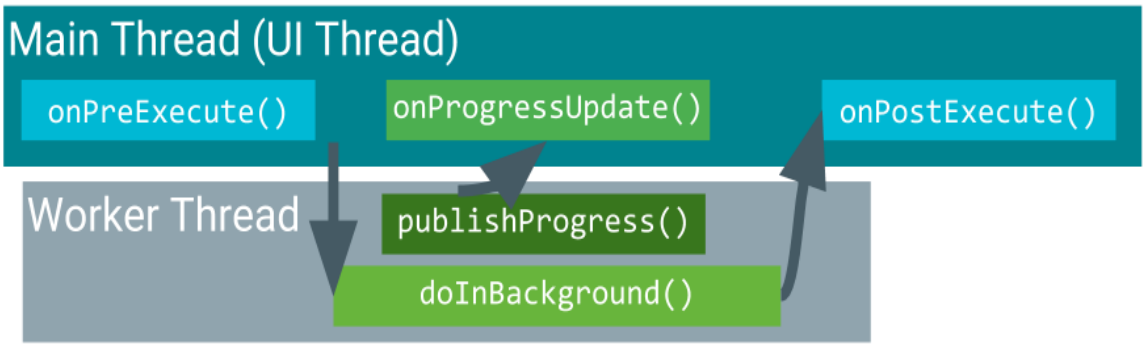
ANS 1:

AsyncTask allows you to perform background operations and publish results on the UI thread without manipulating threads or handlers
AsyncTask is executed, it goes through four steps:

1. onPreExecute() is invoked on the UI thread before the task is executed. This step is normally used to set up the task, for instance by showing a progress bar in the UI.
2. doInBackground(Params...) is invoked on the background thread immediately after onPreExecute() finishes. This step performs a background computation, returns a result, and passes the result to onPostExecute(). The doInBackground() method can also call publishProgress(Progress...) to publish one or more units of progress.
3. onProgressUpdate(Progress...) runs on the UI thread after publishProgress(Progress...) is invoked. Use onProgressUpdate() to report any form of progress to the UI thread while the background computation is executing
4. onPostExecute(Result) runs on the UI thread after the background computation has finished



ANS 2:
Network security and how to make network calls, which involves these tasks:
1. Include permissions in your AndroidManifest.xml file.
   Before your app can make network calls, you need to include a permission in your AndroidManifest.xml file
2. On a worker thread, make an HTTP client connection that connects to the network and downloads (or uploads) data.
   Most network-connected Android apps use HTTP and HTTPS to send and receive data over the network.
3. Parse the results, which are usually in JSON format.
4. Check the state of the network and respond accordingly.
Making network calls can be expensive and slow, especially if the device has little connectivity. Being aware of the network connection state can prevent your app from attempting to make network calls when the network isn't available.

ANS 3:
A notification is a message your app displays to the user outside your application's normal UI. When you tell the system to issue a notification, the notification first appears to the user as an icon in the notification area, on the left side of the status bar.

Creating notifications
You create a notification using the NotificationCompat.Builder class. (Use NotificationCompat for the best backward compatibility. For more information, see Notification compatibility.) The builder classes simplify the creation of complex objects.

Creating

To create a NotificationCompat.Builder , pass the application context to the constructor:

```
NotificationCompat.Builder mBuilder = new NotificationCompat.Builder(this);
```

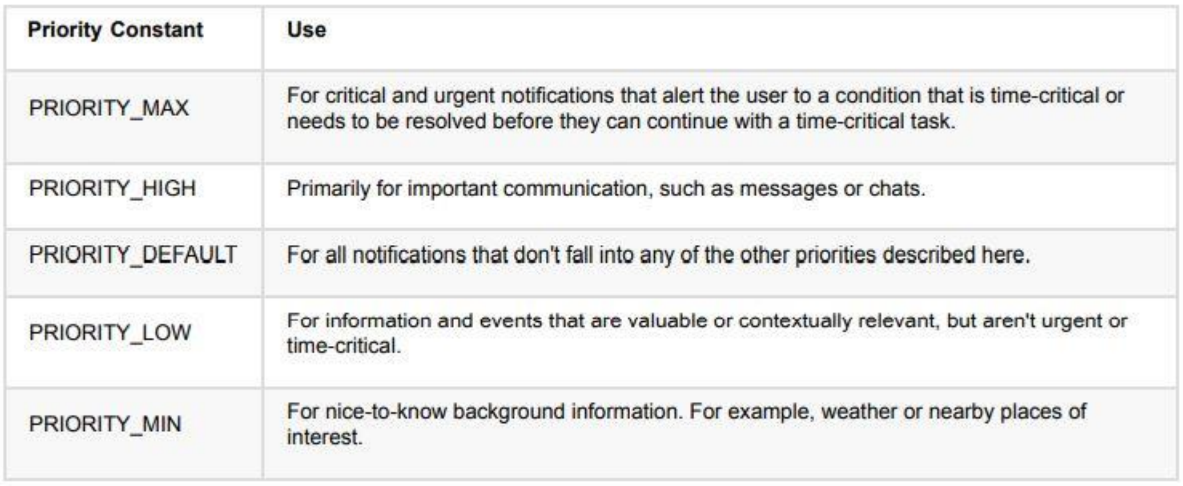
Setting

When using NotificationCompat.Builder, you must assign a small icon, text for a title, and the notification message. You should keep the notification message shorter than 40 characters and not repeat what's in the title. For example:

```
NotificationCompat.Builder mBuilder =

new NotificationCompat.Builder(this)

.setSmallIcon(R.drawable.notification_icon)

.setContentTitle("Dinner is ready!")

.setContentText("Lentil soup, rice pilaf, and cake for dessert.");
```

ANS 4:

Android allows you to assign a priority level to each notification to influence how the Android system will deliver it. Notifications have a priority between MIN (-2) and MAX (2) that corresponds to their importance. The following table shows the available priority constants defined in the Notification class.



Priority Constant	Use
PRIORITY_MAX	For critical and urgent notifications that alert the user to a condition that is time-critical or needs to be resolved before they can continue with a time-critical task.
PRIORITY_HIGH	Primarily for important communication, such as messages or chats.
PRIORITY_DEFAULT	For all notifications that don't fall into any of the other priorities described here.
PRIORITY_LOW	For information and events that are valuable or contextually relevant, but aren't urgent or time-critical.
PRIORITY_MIN	For nice-to-know background information. For example, weather or nearby places of interest.

ANS 5:

Elapsed real-time alarms

Elapsed real-time alarms use the time, in milliseconds, since the device was booted. Elapsed real-time alarms aren't affected by time zones, so they work well for alarms based on the passage of time. For example, use an elapsed real-time alarm for an alarm that fires every half hour.

The AlarmManager class provides two types of elapsed real-time alarm:

- ELAPSED_REALTIME : Fires a PendingIntent based on the amount of time since the device was booted, but doesn't wake the device. The elapsed time includes any time during which the device was asleep. All repeating alarms fire when your device is next awake.
- ELAPSED_REALTIME_WAKEUP : Fires the PendingIntent after the specified length of time has elapsed since device boot, waking the device's CPU if the screen is off. Use this alarm instead of ELAPSED_REALTIME if your app has a time dependency, for example if it has a limited window during which to perform an operation.

Real-time clock (RTC) alarms

Real-time clock (RTC) alarms are clock-based alarms that use Coordinated Universal Time (UTC). Only choose an RTC alarm in these types of situations:

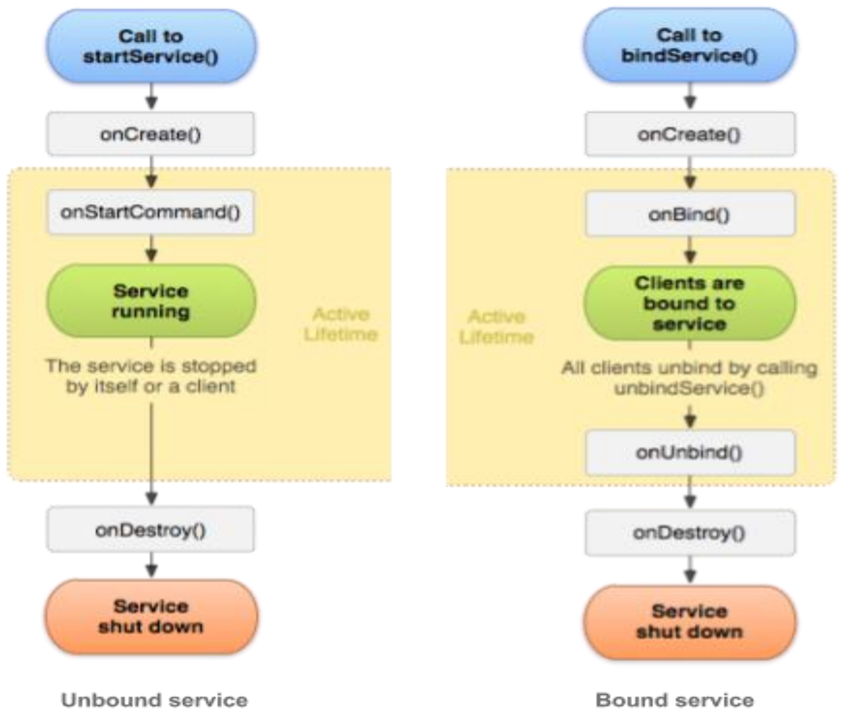
- You need your alarm to fire at a particular time of day.
- The alarm time is dependent on current locale.

Apps with clock-based alarms might not work well across locales, because they might fire at the wrong times. And if the user changes the device's time setting, it could cause unexpected behavior in your app.

The AlarmManager class provides two types of RTC alarm:

- RTC : Fires the pending intent at the specified time but doesn't wake up the device. All repeating alarms fire when your device is next awake.
- RTC_WAKEUP : Fires the pending intent at the specified time, waking the device's CPU if the screen is off.

ANS 6:



A bound service exists only to serve the application component that's bound to it, so when no more components are bound to the service, the system destroys it. Bound services don't need to be explicitly stopped the way started services do (using stopService() or stopSelf())

Bound Services:

A service is "bound" when an application component binds to it by calling bindService() . A bound service offers a clientserver interface that allows components to interact with the service, send requests, and get results, sometimes using interprocess communication (IPC) to send and receive information across processes. A bound service runs only as long as another application component is bound to it. Multiple components can bind to the service at once, but when all of them unbind, the service is destroyed. A bound service generally does not allow components to start it by calling startService() .

Started services:

How a service starts:

1. An application component such as an activity calls startService() and passes in an Intent . The Intent specifies the service and includes any data for the service to use.

2. The system calls the service's onCreate() method and any other appropriate callbacks on the main thread. It's up to the service to implement these callbacks with the appropriate behavior, such as creating a secondary thread in which to work.

3. The system calls the service's onStartCommand() method, passing in the Intent supplied by the client in step 1. (The client in this context is the application component that calls the service.)

ANS 7:

Constantly monitoring the connectivity and battery status of the device can be a challenge, and it requires using components such as broadcast receivers, which can consume system resources even when your app isn't running. Because transferring data efficiently is such a common task, the Android SDK provides a class that makes this much easier: JobScheduler.

JobScheduler has three components:

1. JobInfo uses the builder pattern to set the conditions for the task.

2. JobService is a wrapper around the Service class where the task is actually completed.

3. JobScheduler schedules and cancels tasks.

JobInfo:

Set the job conditions by constructing a JobInfo object using the JobInfo.Builder class. The JobInfo.Builder class is instantiated from a constructor that takes two arguments: a job ID (which can be used to cancel the job), and the ComponentName of the JobService that contains the task. Your JobInfo.Builder must set at least one, non-default condition for the job.

The JobInfo.Builder class has many set() methods that allow you to determine the conditions of the task. Below is a list of available constraints with their respective set() methods and class constants:

- Backoff/Retry policy: Determines when how the task should be rescheduled if it fails. Set this condition using the setBackoffCriteria() method, which takes two arguments: the initial time to wait after the task fails, and the backoff strategy. The backoff strategy argument can be one of two constants: BACKOFF_POLICY_LINEAR or BACKOFF_POLICY_EXPONENTIAL . This defaults to {30 seconds, Exponential}.

- Minimum Latency: The minimum amount of time to wait before completing the task. Set this condition using the setMinimumLatency() method, which takes a single argument: the amount of time to wait in milliseconds.
- Override Deadline: The maximum time to wait before running the task, even if other conditions aren't met. Set this condition using the setOverrideDeadline() method, which is the maximum time to wait in milliseconds.
- Periodic: Repeats the task after a certain amount of time. Set this condition using the setPeriodic() method, passing in the repetition interval. This condition is mutually exclusive with the minimum latency and override deadline conditions: setting setPeriodic() with one of them results in an error.
- Persisted: Sets whether the job is persisted across system reboots. For this condition to work, your app must hold the RECEIVE_BOOT_COMPLETED permission. Set this condition using the setPersisted() method, passing in a boolean that indicates whether or not to persist the task.
- Required Network Type: The kind of network type your job needs. If the network isn't necessary, you don't need to call this function, because the default is NETWORK_TYPE_NONE . Set this condition using the setRequiredNetworkType()
- method, passing in one of the following constants: NETWORK_TYPE_NONE , NETWORK_TYPE_ANY , NETWORK_TYPE_NOT_ROAMING , NETWORK_TYPE_UNMETERED .
- Required Charging State: Whether or not the device needs to be plugged in to run this job. Set this condition using the setRequiresCharging() method, passing in a boolean. The default is false .
- Requires Device Idle: Whether or not the device needs to be in idle mode to run this job. "Idle mode" means that the device isn't in use and hasn't been for some time, as loosely defined by the system. When the device is in idle mode, it's a good time to perform resource-heavy jobs. Set this condition using the setRequiresDeviceIdle() method, passing in a boolean. The default is false .

2. JobService

Once the conditions for a task are met, the framework launches a subclass of JobService , which is where you implement the task itself. The JobService runs on the UI thread, so you need to offload blocking operations to a worker thread.

- onStopJob()
  - The system calls onStopJob() if it determines that you must stop execution of your job even before you've call jobFinished() . This happens if the requirements that you specified when you scheduled the job are no longer met.
- onStartJob()
  - The system calls onStartJob() and automatically passes in a JobParameters object, which the system creates with information about your job. If your task contains long-running operations, offload the work onto a separate thread. The onStartJob() method returns a boolean: true if your task has been offloaded to a separate thread (meaning it might not be completed yet) and false if there is no more work to be done.

Use the jobFinished() method from any thread to tell the system that your task is complete. This method takes two parameters: the JobParameters object that contains information about the task, and a boolean that indicates whether the task needs to be rescheduled, according to the defined backoff policy.

3. **JobScheduler**
   The final part of scheduling a task is to use the JobScheduler class to schedule the job. To obtain an instance of this class, call getSystemService(JOB_SCHEDULER_SERVICE) . Then schedule a job using the schedule() method, passing in the JobInfo object you created with the JobInfo.Builder.
   The framework is intelligent about when you receive callbacks, and it attempts to batch and defer them as much as possible. Typically, if you don't specify a deadline on your job, the system can run it at any time, depending on the current state of the JobScheduler object's internal queue; however, it might be deferred as long as until the next time the device is connected to a power source. To cancel a job, call cancel() , passing in the job ID from the JobInfo.Builder object, or call cancelAll().

ANS 8:

System broadcast intents

The system delivers a system broadcast intent when a system event occurs that might interest your app. For example:

- When the device boots, the system sends an ACTION_BOOT_COMPLETED system broadcast intent. This intent contains the constant value "android.intent.action.BOOT_COMPLETED".
- When the device is connected to external power, the system sends ACTION_POWER_CONNECTED , which contains the constant value "android.intent.action.ACTION_POWER_CONNECTED" . When the device is disconnected from external power, the system sends ACTION_POWER_DISCONNECTED .
- When the device is low on memory, the system sends ACTION_DEVICE_STORAGE_LOW . This intent contains the constant value "android.intent.action.DEVICE_STORAGE_LOW" .

ACTION_DEVICE_STORAGE_LOW is a sticky broadcast, which means that the broadcast value is held in a cache. If you need to know whether your broadcast receiver is processing a value that's in the cache (sticky) or a value that's being broadcast in the present moment, use isInitialStickyBroadcast().

To receive system broadcast intents, you need to create a broadcast receiver.

Custom broadcast intents

Custom broadcast intents are broadcast intents that your application sends out. Use a custom broadcast intent when you want your app to take an action without launching an activity, for example when you want to let other apps know that data has been downloaded to the device and is available for them to use.

To create a custom broadcast intent, create a custom intent action. To deliver a custom broadcast to other apps, pass the intent to sendBroadcast() , sendOrderedBroadcast() , or sendStickyBroadcast().

ANS 9:

Background tasks are commonly used to load data such as forecast reports or movie reviews. Loading data can be memory intensive, and you want the data to be available even if the device configuration changes. For these situations, use loaders, which are a set of classes that facilitate loading data into an activity. Loaders use the LoaderManager class to manage one or more loaders. LoaderManager includes a set of callbacks for when the loader is created, when it's done loading data, and when it's reset.

Starting a loader

Use the LoaderManager class to manage one or more Loader instances within an activity or fragment. Use initLoader() to initialize a loader and make it active. Typically, you do this within the activity's onCreate() method. For example:

getLoaderManager().initLoader(0, null, this);

If you're using the Support Library, make this call using getSupportLoaderManager() instead of getLoaderManager() . For example:

getSupportLoaderManager().initLoader(0, null, this);

The initLoader() method takes three parameters:

- A unique ID that identifies the loader. This ID can be whatever you want.
- Optional arguments to supply to the loader at construction, in the form of a Bundle . If a loader already exists, thisvparameter is ignored.
- A LoaderCallbacks implementation, which the LoaderManager calls to report loader events. In this example, the local class implements the LoaderManager.LoaderCallbacks interface, so it passes a reference to itself, this .

The initLoader() call has two possible outcomes:

- If the loader specified by the ID already exists, the last loader created using that ID is reused.
- If the loader specified by the ID doesn't exist, initLoader() triggers the onCreateLoader() method. This is where you implement the code to instantiate and return a new loader.
- Put the call to initLoader() in onCreate() so that the activity can reconnect to the same loader when the configuration changes. That way, the loader doesn't lose the data it has already loaded.

ANS 10:

In Module 4 qb