4.
What is a service? It is an application component without a graphical user interface. It allows us to make CPU intensive or blocking operations even when the application is not in foreground.

5.
A service can be exported and accessed by external applications.
By default it runs in the main UI thread, so if we want to do CPU intensive or blocking operations we should create a separate thread or configure the service to run in a different process.

6.
Let's see how we declare a service in the Manifest file. We are going to use the <service> tag, which is under the <application> tag.
It has the component android:name, which is the name of the class that implements the service.
Then we have android:enable, if this is false, the system cannot instantiate your service even if other component tries to start your service.

7.
The next tag is very important when we want to export our service, it is called android:exported. If it's configured to be true it tells the system that this service can be accessed from external applications. When we specify an intent filter, the default value is true (service is exported). Without an intent filter, the default value is false (other apps cannot access the service.
When we want to configure a service to run in another process we will use android:process to specify the name of the process.
Then, we have android:permission, which specifies the permissions required by other applications in order to access our service.

8.
We have two types of services: Started Services and Bound Services.
There is not a clear difference between the two, in the sense that a service can be both started and bound in the same time.

The first type of services are called started, because they are launched with the method startService().
A started service is used in general to make a single operation, without returning a result to the caller.
A started service will continue to run even if the caller has terminated. It is not bound to the caller in any way.
An important thing about started services is that they are not killed by the system immediately after they finished their job, if it has enough memory. They will be killed only in low-memory situations.

9.
The second type of services are called bound, because they are started with the method bindService().
Their purpose is to provide a client-server interface. A client component binds to a service, then calls a method of that service and expects a result. They are usually used for the communication between two applications: the client is in one application and the service in another application.
When binding to a service for the first time, an instance is created, and when another app binds to it, it works with the existing instance, it does not create a new instance of the service. If the first component calls unbind, the service is not killed, but it will be killed when the last component calls unbind.
So while a started service runs indefinitely until the system decides to kill it, a bound service is more deterministic, will be killed when the last component calls unbind.

10.
On this slide we have the lifecycle of services (you've seen this image in the second lecture).
When we call startService() or bindService(), the method onCreate() will be called for both types of services. Then, for the two types of services, different methods will be called: for the started services: onStartCommand() and for bound services onBind().
There is a trick here: when you extend the Service class, it is mandatory to implement onBind(). But for started services you will return null.
After these methods are called, both types of services will continue to run.
A Started service will run indefinitely or until another component stops it explicitly with stopService() or the service stops itself with stopSelf().
A Bound service will run while at least one component is bound to it, when the last component calls unbind() it will be killed.
For both types of services, in the end onDestroy() is called.

12.
Now let's see more about the started services.
We will start them by calling startService() from the Context class, and we must give an Intent to this method.
The recommendation is to tell the service what to do by specifying the information in the Intent, and you can do this very easy by setting an extra in the Intent. The service will extract the extra in onStartCommand() and see what job it has to perform. Or you can set the action of the intent and extract the action in onStartCommand().
The service will continue to run even if it has finished the job and can be stopped in two ways: if the component that started it calls stopService() (also passing an Intent) or the service may decide to call stopSelf().

13. How do we implement a started service?
We have two options:

1) the first option is to extend the class called Service and to implement the method onStartCommand(). The method receives an Intent, flags and a startId (which is practically an a unique request id). If we have a long running or blocking operation

(long running means something that takes more than 5 seconds) we should create a separate thread manually. Because when we extend the Service class, the service will run on the main UI thread.

When onStartCommand() is executed in the service you can return START_STICKY or START_NOT_STICKY. If you return START_NOT_STICKY, this means that the service should not stick around, so the service will be killed faster when the system has low memory and will restart it less quickly. If you return START_STICKY, the system will want to restart the service as soon as possible, if it gets killed.
We also have to implement onBind() because it is mandatory, but will return null in a started service.

14.
This is an example with a class extending Service.
In the first box we have a partial implementation of a started service. In onStartCommand() we receive an intent, which may contain extras that indicate the action that must be executed by the service. The method returns START_STICKY, so the system will restart the service if it gets killed.

15.

2) the second option is extend IntentService, which is a class that extends Service. The advantage is that is has its own worker thread which deals with requests. The disadvantage is that requests are serialized on this thread. So, if two requests arrive at the service, they will be serialized. In the previous option you can create a thread for each request. When we extend IntentService we only have to implement the method onHandleIntent().

16.
An IntentService practically creates a worker thread that handles all Intents that are received onStartCommand(). This is a separate thread from the main UI thread of the application.
It also creates a work queue that delivers one Intent at a time to the implementation of onHandleIntent(). This is how it serializes requests.
It destroys the service after all requests have been treated, so you don’t have to call stopSelf() explicitly.
IntentService includes a default implementation of onBind() that returns null.
It also includes an implementation of onStartCommand() that takes every Intent and sends it to the work queue and then to the implementation of onHandleIntent().

17.
This is an example with a class extending IntentService.
Here, you just need to implement onHandleIntent() in order to obtain the information from the intent and perform the necessary operations. The method runs on a worker thread, so there is no need to create another thread to handle the request.
If you want to override other callback methods, you should not forget to call the super implementation. This way, the IntentService handles the life of the worker thread correctly.

18.
You can start a service from an activity or other application component by passing an Intent to startService().
If the service is not already running, the Android system will call the service's onCreate() and then onStartCommand() and pass the Intent.
If the service is already running, the system will call only onStartCommand() and pass the Intent.
In the box, we can see how the service is actually started from another component. An intent is created by specifying the class name of the service.
Then we call startService by giving it the intent.

19.
A started service must handle its own lifecycle. After a service is started, it will continue to run and will not be killed by the system excepting for the low memory situations.
Therefore, you should explicitly destroy a started service either by calling stopSelf() from the service or by calling stopService() from other application component.
However, you should pay attention not to kill the service while it is still processing requests from other components. You should ensure that the request to stop the service is based on the most recent start request by using the startId parameter that is received by stopSelf(). If there is a more recent request being processed, the IDs will not match and the service will not stop.

21.
Now we are moving on to the Bound Services.
These are started when a component calls bindService() from the Context class (and giving it an Intent). If the service was stopped, and someone calls bindService(), it will be started, will go through onCreate() and onBind(). The second component that calls bindService() receives directly the instance that is already created, and the service does not go through the whole lifecycle again.
Bound services are based on a client-server paradigm, in which the server is the service and the client is an external component, usually an activity which calls the methods of the service.
The whole communication is not done through sockets, but through the Binder. More exactly, in the SDK, in the Java part, the IBinder interface are used.
With a bound service we can talk from external applications. And the question is how do we want to manage the requests that arrive in the same time? Do we want to treat them simultaneously or we can serialize them?

22.
Let's see how we can implement a bound service.
First of all we need to extend the Service class.
Then we must implement the onBind() method that needs to return an IBinder object. This method will be called only once when the first component calls bindService(). The second component that calls bindService() will receive directly the same IBinder object, but onBind() will not be executed again.

23.
If the client is running in the same process, you can extend the Binder class and return it to the caller. But this will not work if the client runs in a different process.
For the communication with a client that runs in a separate process (in the same or different application), you have two options:

1) To use a Messenger and call Messenger.getBinder(). This method serializes all incoming requests, so it will not handle them simultaneously
2) The second method is to use AIDL, which is better when you want to handle requests simultaneously

24.
On the client side, connecting to a bound service is done by implementing the ServiceConnection interface. Here you have to implement two callbacks: onServiceConnected() and onServiceDisconnected().
In the onServiceConnected() you receive an IBinder instance that will be used for communicating with the service.
You may also need to use onServiceDisconnected() because the service may be killed. For example, if the service is both started and bound, if a component called stopService(), the service will be killed even if there was another component bound to it. In that moment onServiceDisconnected() will be called and you will find out that the service has been killed. This is where you should implement what happens when the service dies for some reason.

25.
After we have a ServiceConnection, we must call the method bindService() and give it the ServiceConnection as argument. The method bindService() returns immediately, it does not block until the connection is established. We can find out when the connection is ready when onServiceConnected() is called by the system.

26.
To close the connection with a service we call unbindService(), and if it is the last component bound to the service, it will be killed. However, if there is another component that “started” the service, then it will not be killed.

27.
First of all, let’s see what we have to do if the service and client run in the same process.
In the service, we should create a member variable that extends the Binder class.
Then, from the service’s onBind() method, return this member variable.

28.
For exporting services, when the client runs in the same process with the service we have 3 options:

1) The first option is to for the class that extends Binder to have public methods. So the application that receives the Binder instance to make cast to our class and then to start calling methods from it.

2) The second option is for the class that extends Binder to have a method called getService() that returns an instance of the service. And in the client, you get the Binder, make a cast to our class and then call getService(). Then you can call directly the public methods of the service.
3) The third option is to have a third class in the service, in which we can define an interface for the operations that can be performed by the service. Then in the Binder instance, we have a method that returns an instance of that third class and we can use that class as communication interface between the client and the service.

29.
Here we have an example of returning a reference to the service class. This works only if the client and the service run in the same process.
So we have a class LocalService that extends Service.
We implemented a class LocalBinder that extends Binder and has a method getService() that returns the instance of the service (LocalService). And in the onBind() we return an instance of the LocalBinder.
The service has one public method called getRandomNumber().

30.
Here we have part of the client code, which is an activity. We can see that in onStart(), we call bindService() and in onStop() we call unbindService().
We have a ServiceConnection instance called mConnection, and we implement the method onServiceConnected() and onServiceDisconnected().
In onServiceConnected() we receive an IBinder and cast it to LocalBinder. Then we obtain the instance of the service using the method getService() of the LocalBinder.
After that, we can use the public methods of the service directly, such as getRandomNumber() which is a public method from the service.

32.
Now we are talking about the communication between bound services and external applications.
One option is to communicate through a Messenger. It is a class that helps you perform remote IPC with another application or a component from the current application. This class will serialize multiple incoming connections.

33.
How do we use a Messenger?
In the Service class, we must extend the Handler class, in which you must implement the method handleMessage(). We decide what operation to perform depending on the received message. In each message you can have a bundle of data, and depending on what you have in that bundle you can make one operation or another.

After extending the Handler class, we must create a Messenger instance and in the constructor we give it the Handler instance. In onBind() we call the method Messenger.getBinder() and return that Binder.

34.
Here we have an example of a service using a Messenger.
In our service class, we implement IncommingHandler which extends the Handler class. It implements handleMessage(). If the message is MSG_SAY_HELLO, then it displays a toast in the screen.
Then we create a Messenger instance and give the Handler instance to the constructor.
And in onBind() we return the Binder obtained through Messenger.getBinder().

35.
On the client side, in onServiceConnected(), we can create a Messenger object based on the IBinder object received in onServiceConnected().
And then we can create messages and send them through that Messenger.

36.
This is an example of a client, in the ServiceConnection instance, in onServiceConnected(), we obtain a Messenger instance, by giving the IBinder to the constructor.
When we want to send a message to the service, we make a Message object and send it through the send() method of the Messenger.

37.
What is tricky here is that the handleMessage() method returns void. This means that we won't have a two way communication with the service. You will be able to send messages to the service in order to determine the execution of several operations, but the service will not be able to respond.
If we want a two way communication we must implement a similar Messenger mechanism on the client side. The client must have a Messenger class. Every message, has a parameter called reply to. So in onServiceConnected we receive a Message instance and we set the reply to of that Message to the Messenger of the client. When we send that Message to the service, and when handleMessage() will be called in the service, it can take that reply to parameter, which is the Messenger class of the client, so it can call methods from the client.
The communication through Messengers is a little tricky, that is why I prefer the second method, through AIDL.

39.
IDL comes from Interface Definition/Description Language. Both are accepted.
It was developed with the following reason: we have two components, one written in C and one in Java, and they should be able to talk to each other.
How do we make this happen? They developed a third language, that will not be a programming language, but a specification language. So we can specify how the communication between two entities looks like.

IDLs are used whenever we are talking about Remote Procedure Calls because it gives you a method for specifying the communication protocol between two entities.

40.
Examples are: AIDL (Android IDL), one with a funny name: OMG IDL which is based on Corba (you may have learned about Corba in your 4th year), Protocol Buffers from Google is a very used IDL (it is used in the Google APIs as a method for serializing structured data in order to exchange data between two entities, and you have it for Java, C++, Ruby, JavaScript, etc. ), WSDL, which is pretty old but still used, for web services.

41.
Let's see why we need AIDL. On Android we have security through sandboxing, so an application will not have access to the memory of another application.
But there are cases in which we want two applications to communicate with each other, so we need a way to send messages or perform remote procedure calls. We need a way to transform objects into primitives that can be marshalled across the system. This is done through the Binder framework.
The idea was to expose this Binder framework so that it can be used by Android applications written in Java. This is where AIDL comes into discussion. When using AIDL, the system takes care of marshalling and unmarshalling and calling the Binder services.

42.
So this is how it works: we are making an aidl file in the src/ directory of the application that hosts the service. In this file, we define an interface that contains method signatures (the language is similar to Java, with some keywords in addition, for example in, out or inout). The language is very restrictive, because we can use data types (as method parameters and return values) such as primitives (int, char, double, long, and so on), strings, char sequences, list and map (and the system will use ArrayList and HashMap for those, so don't assume that the Map will be sorted). Even if we are using List or Map, we can parametrize them only with other permitted data types, so for example you can have a List of ints or strings, and similar for Map.

43.
So, we have this file, and when we build the application through the IDE, it will generate into the gen/ directory, a file with the same name and the extension java. So, if you had YourInterface.aidl, you will obtain YourInterface.Java.
It will also generate another file called YourInterface.stub, this is a subclass, and it is very important because this class will be implemented in the Service.
So we are going into the code of the Service and implement the class YourInterface.Stub, practically we have to implement the methods described in the aidl file.
So if your aidl had the method int add(int x, int y), when you implement new YourInterface.stub you have to implement this method.
After doing this, in the onBind, we will return this instance that extends IBinder. And that's about it on the service side.

44.

This is an example. We implement the IRemoteService.Stub class in which we implement the methods specified in the aidl file. We return the instance of this class in the onBind() method.

45.
On the client side it's a little more tricky, because you have to take the aidl file and copy it in the other application, so in the src/ folder. What is important here is to have the same package name with the one of the application that includes the service. So if the application that included the service had the package name com.example.myapp, and the client application has the package name com.example.anotherapp, we must create the package com.example.myapp in the second application. This is because the first line in the aidl file contains the package definition.
Then, in the client app, we have a ServiceConnection, and in the onServiceConnected() we receive the IBinder, and having the YourInterface.stub (at build time), we can obtain a reference to the aidl interface through the method YourInterface.Stub.asInterface(IBinder).
It is very important that when you call methods from the service to guard your code with try-catch block, because you need to catch the DeadObjectException. This is an exception that extends RemoteExecution exception. This exception is thrown when the service gets killed by the system because it has low memory.

46.
Then from the client, in the method onServiceConnected, we obtain the instance using IRemoteService.Stub.asInterface() method by giving it the IBinder instance.
Then we can call the methods defined in the mIRemoteService (the ones defined in the aidl file).

47.
I was saying that in an AIDL file you can have only certain types of data: primitives, strings, char sequence, list and map. However, there is a method through which you can have your own class and send it through the Binder. For this you must have a class that implements the interface Parcelable, and implement the methods writeToParcel() and readFromParcel() in order to perform the marshaling and unmarshaling of the data. For example, if you have a an object called Triangle, you can write the coordinates in a certain order in writeToParcel() and read them in the same order in readFromParcel().
This class must also have public static final Parcelable.Creator<YourClass> member variable named CREATOR. For this you must implement createFromParcel() which creates an object from a parcel, and the method newArray(), which returns an object array, from a parcel, this is used when you want to send a list of objects.
After this, you must create an aidl file, with a single line: parcelable YourClass; and that's it. From this moment the system will know how to marshal and unmarshal this class over a Binder.

48.
This is an example of a custom class that implements the Parcelable interface. We implement the method writeToParcel(). And include the member variable CREATOR that includes the methods createFromParcel() and newArray().

50.
Moving on to foreground services. This is when the user is aware of the service (for example playing music). A service that has a certain visual or audio impact over the user. A music player is the best example because it has to play music while the user is using other applications. So the user is aware that another app does something because he hears the music.
And usually, we don’t want these kind of services to be easily killed by the system in low memory situations.
First of all, we will create a notification from the service. You cannot be a foreground service if you don’t show something visual to the user.

51.
After having a notification, we must call startForeground() and to give the Notification as an argument. This announces the system that the service is important and should not be easily killed and also setting the notification that will be shown to the user.
This is called from the service itself. When creating the Notification you have to specify which Activity will be started when clicking on the notification.
When we want to stop that service, we call stopForeground().

52.
This is an example on how to make a service to be foreground service.
First, we create a notification, then we create an intent to the activity that has to be started when clicking on the notification. We create a PendingIntent based on that Intent and associate it to the notification. Finally, we call startForeground().