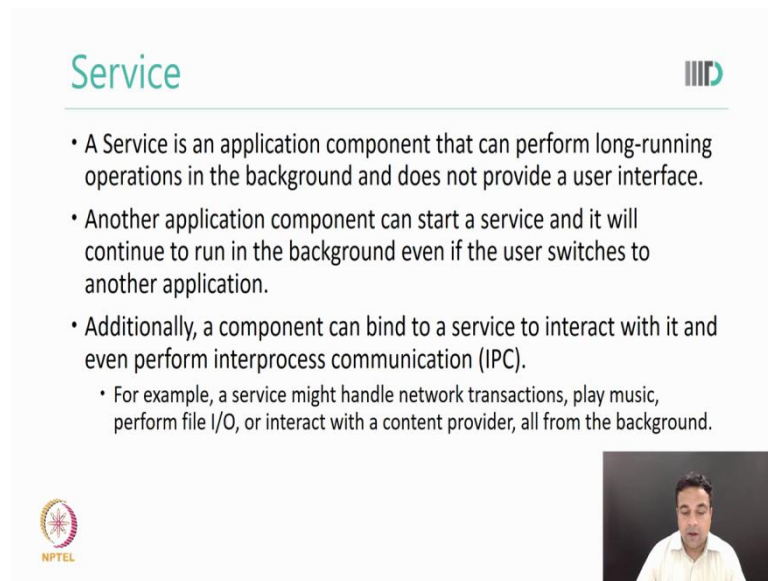


**Mobile Computing**  
**Professor Pushpendra Singh**  
**Indraprastha Institute of Information Technology Delhi**  
**Lecture 31**  
**Services**

Hello, welcome to your new class. Today we will learn about services. As explained to you earlier services are one of the 4 app components. So far we have studied activities in detail, today we will discuss and study services, so let us get started.

(Refer Slide Time: 0:43)



**Service**

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

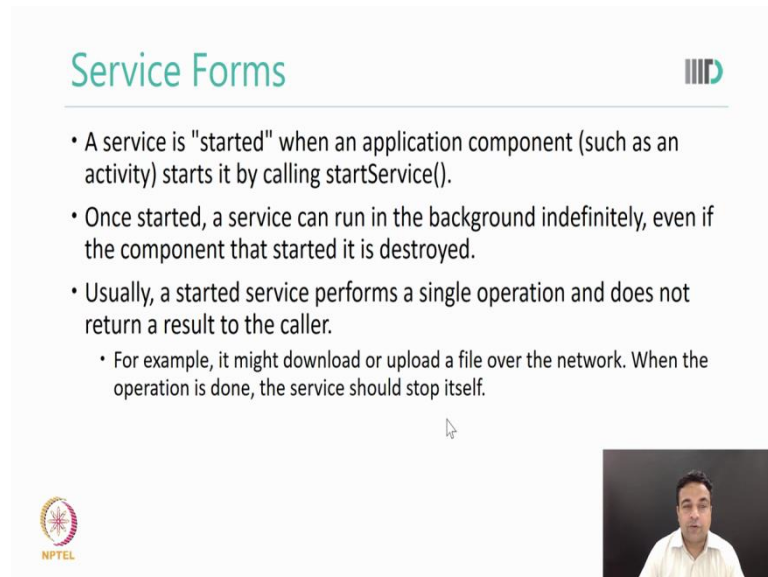
NPTEL

So a service is an application component that can perform long-running operations in the background and does not provide a user interface, so this is the major difference between a service and an activity, activity provides the user interface service does not provide the user interface. So services are useful for example when you want to download a file. Now just like activities another application component can start a service and once the service is started it will continue to run in the background even if the user switches to another application. This is another difference then the activity.

Now additionally on top a component can bind to a service to interact with it and even perform interprocess communication. We will discuss this in detail in some another class but here the service a very briefly let us discuss here the service sort of is providing as a interprocess communication. That is making two processes talk to each other, so you bind to a service and then you can interact with the service. So for example a service might handle network transactions, play music, perform file I/O, or interact with a content provider, all

from the background. So services are very very useful component for an Android applications and today we are going to understand how they work.

(Refer Slide Time: 2:33)



The slide is titled "Service Forms" in a teal font at the top left. In the top right corner, there is a logo consisting of three vertical bars of increasing height followed by a stylized 'D'. The main content of the slide is a list of three bullet points:

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


At the bottom left of the slide is the NPTEL logo, which is a circular emblem with a star in the center and the word "NPTEL" below it. At the bottom right, there is a small video inset showing a man with dark hair wearing a light-colored shirt, speaking.

Now there are primarily two types of service forms, so the first is that what we call as started. So a service is started when an application component such as an activity or another service starts it by calling `startService()`. So once started a service can run in the background indefinitely even if the component that started it is destroyed. That is the life cycle of the service is different than the life cycle of the component. If you remember earlier when we were discussing fragment and activities, fragment dependent on activity.



So a fragment was very decided by the activity. But services this is not the case, even though a service is started by an application component, the service may live even beyond then what the component that started it. So usually a started service performs a single operation and does not return a result to the caller. The single operation could be downloading or uploading a file, it could be playing the music. Now ideally you should design a service in such a way that once the operations complete the service should stop itself, we will discuss it very soon.

(Refer Slide Time: 4:04)

## Service Forms



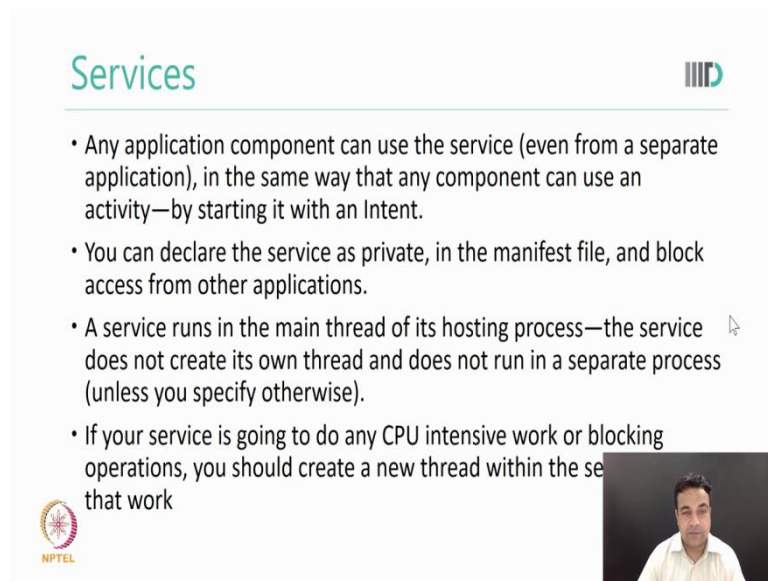
- A service is "bound" when an application component binds to it by calling `bindService()`.
- A bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication (IPC).
- A bound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



Another form of services is what we call a service is “bound”. A service is bound when an application component binds to it by calling a special method named as bind service. Now as discussed earlier a bound service offers a client-server interface that allows components to interact with the service, send requests, get results, and even do so across processes with interprocess communication. So a bound service (()) (04:37) provides you a nice set of services around the IPC. And bound service unlike the started service runs only as long as another application component is bound to it, so if there is no application component bound to it the service is destroyed. So this is different than started service, where it is not destroyed even when the component which is started it has been destroyed.

And bound service the movement no other application component is bound the service also get destroyed. And multiple components can bind to the service at once however at least one component must be bounded to the service in order for the service to left soon. As long, so as soon as there is no component bound to a service is destroyed.

(Refer Slide Time: 5:41)



The slide is titled "Services" in a teal font. It contains a list of four bullet points. In the bottom right corner, there is a small video inset showing a man in a light-colored shirt. The NPTEL logo is in the bottom left corner.

- Any application component can use the service (even from a separate application), in the same way that any component can use an activity—by starting it with an Intent.
- You can declare the service as private, in the manifest file, and block access from other applications.
- A service runs in the main thread of its hosting process—the service does not create its own thread and does not run in a separate process (unless you specify otherwise).
- If your service is going to do any CPU intensive work or blocking operations, you should create a new thread within the service that work

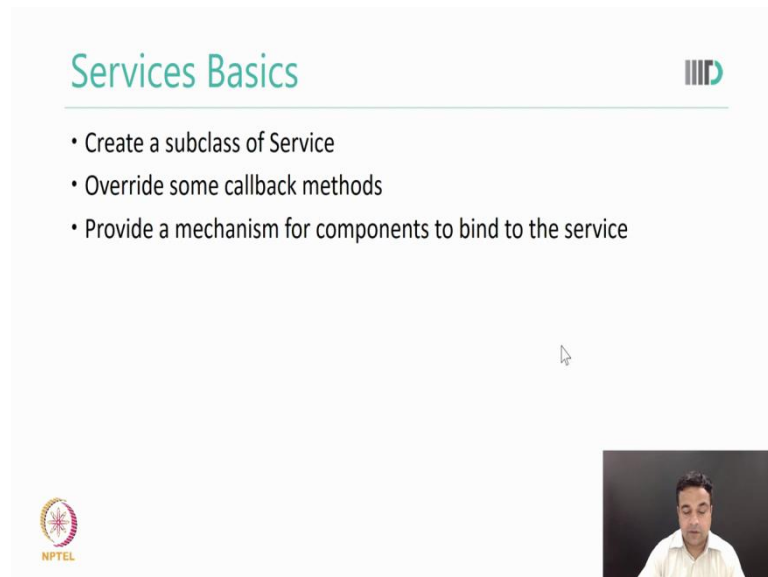
Now let us discuss something more about services. So any application component can use the service just as we discussed earlier. Earlier, in previous lectures we were told what are the application components these are primarily activity, services, background receivers and content providers. So any application component can use the service even from separate applications similar to an activity in the same way that any component can use an activity. And in fact the method of starting a service is also very similar to the method that we learn when we wanted to start an activity. That is we can start a service by intent. So we can use an intent and we can start the service. Now you can declare a service as private in the manifest file and you can block access from other applications, so this is one difference between services and applications.

In applications also you can try to control it by using intent filters, etc, but service you can declare as private. So a service runs in the main thread of its hosting process. And the service does not create its own thread and does not run in a separate process, so what does it mean. So very soon in the next lecture we will learn more about processes and threads but if you go back to your operating system knowledge then usually there is a process running on its own in the main thread and unless you start a separate thread everything runs in the main thread.

Now this makes the execution more predictable and more understandable however, if there is a long running task then the specially for the UI type of work the UI suffers. So similarly, when we are creating a service here the service will run in the main thread of its hosting process and does not create its own threads so this is something to remember. So if your service is going to do any CPU intensive work or blocking operation, you should create a

new thread within the service to do that work, if you do not do it service will keep running in the main thread and then the main thread will get occupied and the activity is of the service and that may hurt the experience of the user and that may hurt the performance of your application.

(Refer Slide Time: 8:22)



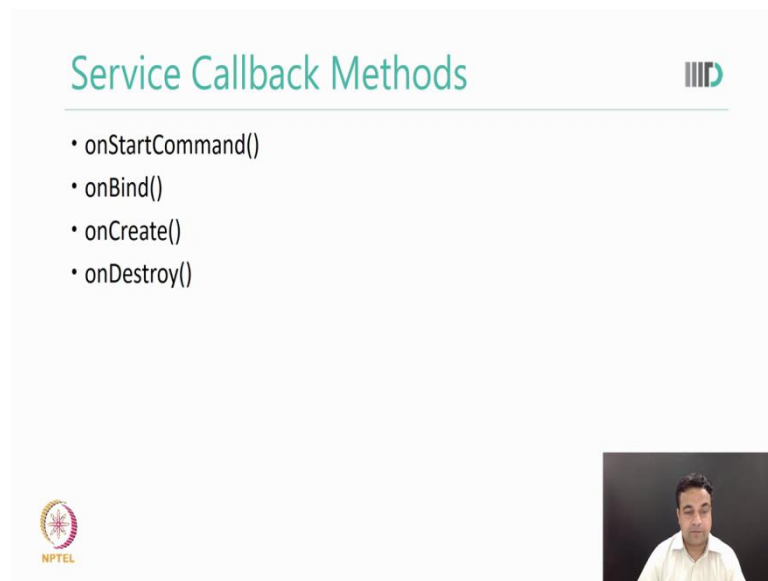
**Services Basics**

- Create a subclass of Service
- Override some callback methods
- Provide a mechanism for components to bind to the service

NPTEL

Now let us get started with the basics of the service, so what do we need to do if we want to let us say create a service. So we learned how to create an activity, we learned how to create a fragment, now we are trying to learn how to create a service. So creating a service is also very easy, we create a subclass of service, you override some call-back methods just like in case of activities in fragments and then for the bound service for the bound services only we also provides a mechanism for components to bind to the service. So by doing these three operations or mainly two operations you can start a service. Now let us have a look at the service call back methods and we will only look at the important call back methods.

(Refer Slide Time: 9:15)



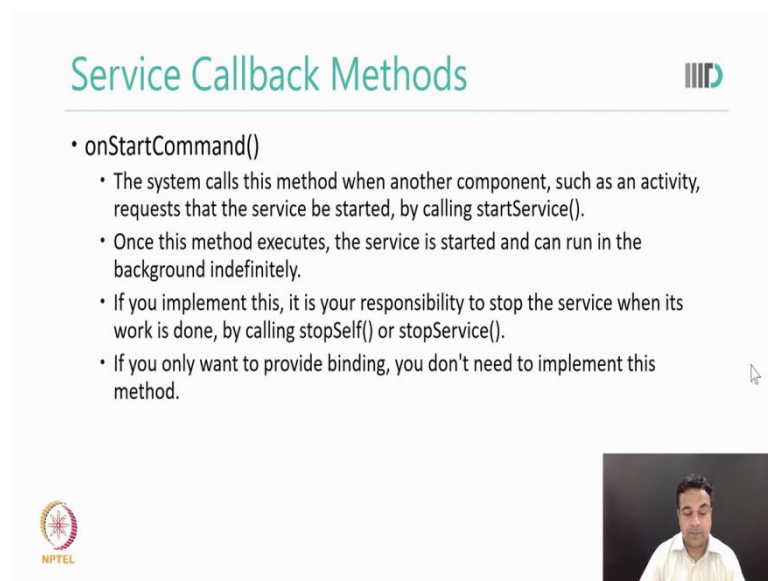
## Service Callback Methods

- `onStartCommand()`
- `onBind()`
- `onCreate()`
- `onDestroy()`

NPTEL

So there are 4 important callback methods for service `onStartCommand()`, `onBind()`, `onCreate()` and `onDestroy()`. Now let us get started with the first method the `onStartCommand()`.

(Refer Slide Time: 9:35)



## Service Callback Methods

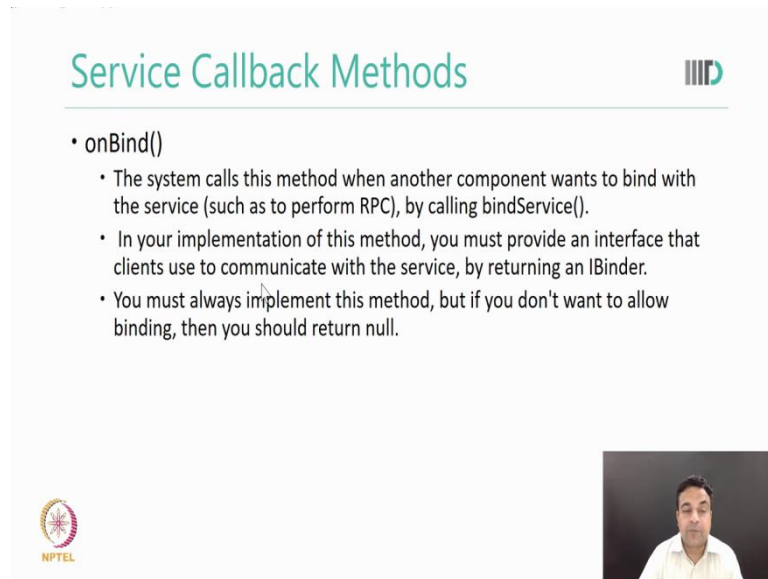
- `onStartCommand()`
  - The system calls this method when another component, such as an activity, requests that the service be started, by calling `startService()`.
  - Once this method executes, the service is started and can run in the background indefinitely.
  - If you implement this, it is your responsibility to stop the service when its work is done, by calling `stopSelf()` or `stopService()`.
  - If you only want to provide binding, you don't need to implement this method.

NPTEL

Now the system calls this method when another component such as an activity request that the service we started by calling start service. Now please try to pause yourself here and think that what happens in case of activity and you will find some similarities. Now once this method executes the services is started and can run in the background indefinitely. So if you are implementing this method it becomes your responsibility that is the responsibility of the developer to stop the service when the work is done by either calling `stopSelf()` or by calling

stopService(). Now if you only want to provide binding and not the started services we discussed earlier we do not need to implement this method. Now let us move on to the second method.

(Refer Slide Time: 10:49)



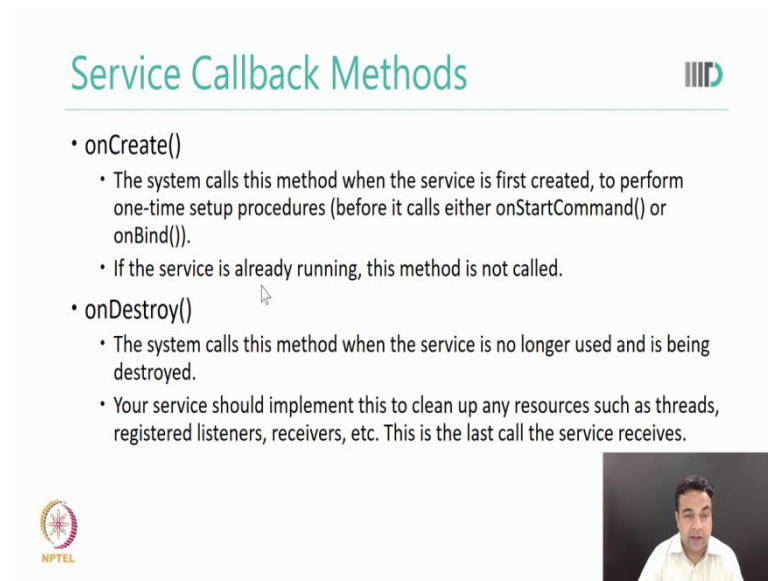
The slide is titled "Service Callback Methods" in a teal font. It features a list of details for the `onBind()` method. In the bottom right corner, there is a small video inset showing a man in a light blue shirt. The NPTEL logo is in the bottom left corner.

## Service Callback Methods

- `onBind()`
  - The system calls this method when another component wants to bind with the service (such as to perform RPC), by calling `bindService()`.
  - In your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an `IBinder`.
  - You must always implement this method, but if you don't want to allow binding, then you should return null.

The second important method is the `onBind()` method. So the system calls this method when another component wants to bind with the service (such as to perform RPC), by calling `bindService()`. Now in your implementation of this method, you must provide an interface that clients use to communicate with the service, by returning an `IBinder`. And you must also implements this method, but if you do not want to allow binding, then you should return the null. So we always implement `onBind()` if you want a bounded service, we return the correct type otherwise we just return null.

(Refer Slide Time: 11:34)



The slide is titled "Service Callback Methods" in a teal font. It features a list of two methods: `onCreate()` and `onDestroy()`. Each method has a bulleted list of details. In the bottom left corner, there is a small NPTEL logo. In the bottom right corner, there is a small video inset showing a man in a white shirt speaking.

## Service Callback Methods

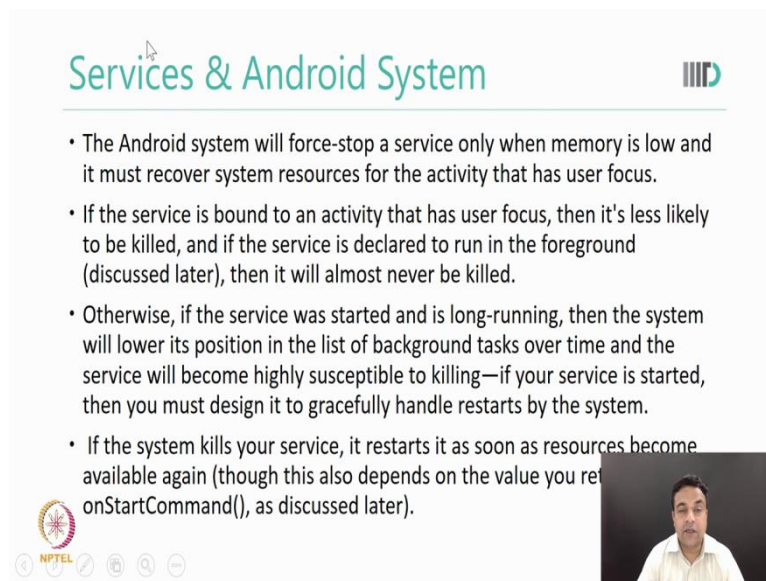
- `onCreate()`
  - The system calls this method when the service is first created, to perform one-time setup procedures (before it calls either `onStartCommand()` or `onBind()`).
  - If the service is already running, this method is not called.
- `onDestroy()`
  - The system calls this method when the service is no longer used and is being destroyed.
  - Your service should implement this to clean up any resources such as threads, registered listeners, receivers, etc. This is the last call the service receives.

Then there is this `onCreate()` method which is kind a similar to the to our earlier studied `onCreate()` methods. Now the system calls this method when the service is first created, so this is same as that was the case with the activity to perform one time setup procedures. And before the system calls either `onStartCommand` or `onBinds()`. So the on `onCreate` is the first method that is called by the system when the service is first created. Now if the service is already running that is some other app component has started the service, then obviously this method is not complete.

The fourth method is the `onDestroy()` method. The system calls this method when the service is no longer used and is being destroyed. So your service should implement to clean up any resources such as threads, registered listeners, receivers, etc, when you get the call to the `onDestroy()`. Now if a component has started a service by calling `start service` then the service remains running until it stops itself with either `stopSelf()` or another component stops it by calling `stop services`. And if a component calls `bind service` to create the service, then the service runs only as long as the component is bound to it or any component is bound to it. So once the service is unbound from all clients then the system destroys itself.



(Refer Slide Time: 13:18)



The slide is titled "Services & Android System" in a teal font. It features a list of four bullet points explaining how the Android system manages services. In the bottom right corner, there is a small video inset showing a man speaking. The bottom left corner contains the NPTEL logo and several navigation icons.

- The Android system will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus.
- If the service is bound to an activity that has user focus, then it's less likely to be killed, and if the service is declared to run in the foreground (discussed later), then it will almost never be killed.
- Otherwise, if the service was started and is long-running, then the system will lower its position in the list of background tasks over time and the service will become highly susceptible to killing—if your service is started, then you must design it to gracefully handle restarts by the system.
- If the system kills your service, it restarts it as soon as resources become available again (though this also depends on the value you return from `onStartCommand()`, as discussed later).



Now Android system itself will force-stop a service only when memory is low and it must recover system resources for the activity that has user focus. Now try to remember here the activity has a user interface, service does not have a user interface. So if the memory is low and the system must recover system resources the Android system will force stop a service in order to keep alive an activity. Now if the service is bound to an activity that has user focus, then obviously it is less likely to be killed and if the service is declared to run in the foreground then it will almost never be killed.

So we will very soon come on the foreground services, but you can understand that if the service is bound to an activity that has the focus there is no advantage of killing it. Otherwise, if the service was started and is long running, then the system will lower its position in the list of background tasks over time that is kind of reduce the priority and service will become highly susceptible to killing. That is if the system needs resources it will kill the service. So it is your responsibility that if the service is started then you must design it to gracefully handle restarts by the system. Now if the system kills your service, it restarts it as soon as resources become available again there is a slight catch here because it also depends on the value you return from `onStartCommand()` but we will discuss it very soon

(Refer Slide Time: 15:45)

## Declaring a service in the manifest

```
<manifest ... >  
...  
<application ... >  
  <service android:name=".ExampleService" />  
  ...  
</application>  
</manifest>
```




Now this was all about basics of service and as you may have guessed that services have many thing in common with activity as far as the concepts are concerned and the major difference between service and activity is that while activity has a user interface the service will does not have a user interface. Now let us see that how can we work with services that is how do we write programs that uses services. So the first step just like last time for activities you declare a service in the manifest. Here is an example of a very basic manifest file it starts there is an application tag and inside that application tag I have a service with the name of ExampleService that is it my manifest will include as we know all the app components, so my manifest will also include the services that my application has.

(Refer Slide Time: 16:20)

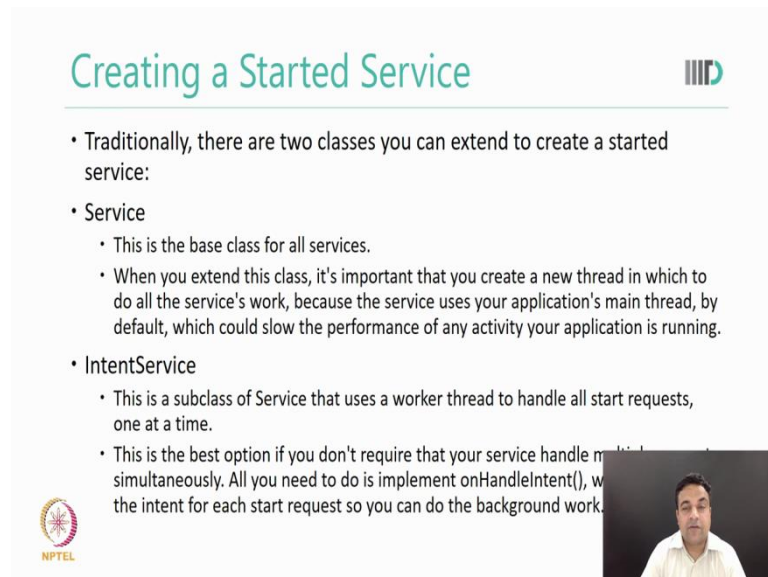
## Creating a Started Service

- A started service is one that another component starts by calling `startService()`, resulting in a call to the service's `onStartCommand()` method.
- The service should stop itself when its job is done by calling `stopSelf()`, or another component can stop it by calling `stopService()`.



Now in order to create a started service as we discussed is the one that another component starts by calling `startService()` and this call to the `startService()` results in a call to the service `onStartCommand()` method. In case of started service, the service must stop itself when its job is done by calling `stopSelf()`, or another component can stop it by calling `stopService()`.

(Refer Slide Time: 17:03)



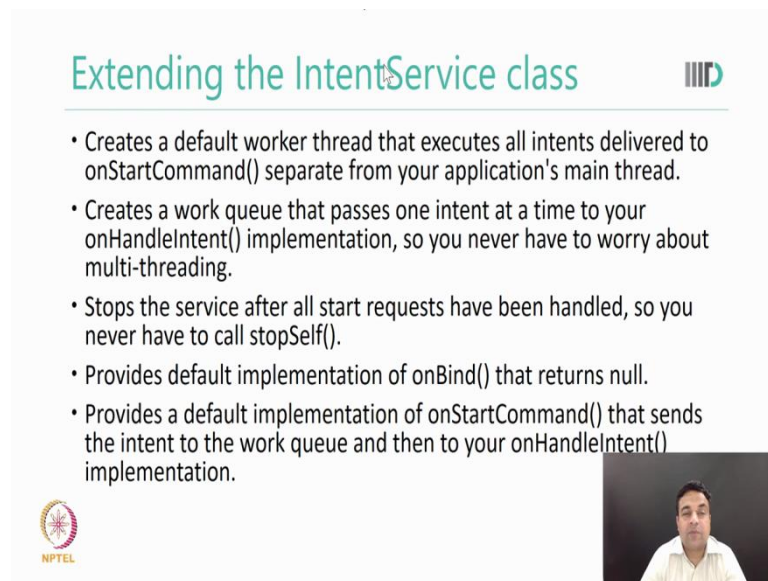
The slide is titled "Creating a Started Service" in a teal font. It features a list of bullet points explaining how to create a started service. The points are: 1. Traditionally, there are two classes you can extend to create a started service: 2. Service: This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running. 3. IntentService: This is a subclass of Service that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement `onHandleIntent()`, which receives the intent for each start request so you can do the background work. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- Traditionally, there are two classes you can extend to create a started service:
- Service
  - This is the base class for all services.
  - When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.
- IntentService
  - This is a subclass of Service that uses a worker thread to handle all start requests, one at a time.
  - This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement `onHandleIntent()`, which receives the intent for each start request so you can do the background work.

Now for creating a started service traditionally there are two classes you can extend to create a started service. The first class is the service class itself which is the base class for all services. And when you extend this class it is important that you create a new thread in which to do all the service work. As I explained earlier that service runs in the main thread and if the service is doing CP intensive work then it affects the UI. So when you extend the service class you create a new thread in which to do all the service work. Now the second method is to by extending intent service, so intent service is a subclass of service that uses a worker thread to handle all start requests one at a time. So the service was the base class and the intent service is a subclass. In service it was in the main thread, in intent service it is using a worker thread.

Now the intent service is the best option if you do not require your service to handle multiple requests simultaneously. Means your service is of the type where you know that there will not be scenario when you will have multiple request simultaneously in that case intent service is much better and easier way to do this. So all you need to do will be to increment the `onHandleIntent()`, which receives the intent for each start request so you can do the background work.

(Refer Slide Time: 18:57)



### Extending the IntentService class

- Creates a default worker thread that executes all intents delivered to `onStartCommand()` separate from your application's main thread.
- Creates a work queue that passes one intent at a time to your `onHandleIntent()` implementation, so you never have to worry about multi-threading.
- Stops the service after all start requests have been handled, so you never have to call `stopSelf()`.
- Provides default implementation of `onBind()` that returns null.
- Provides a default implementation of `onStartCommand()` that sends the intent to the work queue and then to your `onHandleIntent()` implementation.



The slide includes the NPTEL logo in the bottom left and a small video inset of a presenter in the bottom right.

Now let us look at how we first create a service by extending the intent service class. This is an easier method very easy to understand, very easy to implement. So what do we do create a default worker thread that executes all intents delivered to `onStartCommand()` separate from your applications main thread. Another thing that it does it to create a work queue that passes one intent at a time to your `onHandleIntent()` implementation, so that you do not have to worry about the multi thread. So intent service is already doing a lot and then it stops the service after all start requests have been handled, so you will not have to call `stopSelf()`. Intent service also provides default implementation of `onBind()` that return null. If you remember, there was a condition we do not bind that we must override in even if we have to return null it looks like that intent service already does it.

(Refer Slide Time: 20:40)

# Example



```
public class HelloIntentService extends IntentService {  
  
    /**  
     * A constructor is required, and must call the super IntentService\(String\)  
     * constructor with a name for the worker thread.  
     */  
    public HelloIntentService() {  
        super("HelloIntentService");  
    }  
  
    /**  
     * The IntentService calls this method from the default worker thread with  
     * the intent that started the service. When this method returns, IntentService
```



And then intent service also provides a default implementation of `onStartCommand()` that sends the intent to the work queue and then to your `onHandleIntent()` implementation. Here is a simple example of the intent service extension to create a service. So let me start from the very beginning. So there is a public class `HelloIntentService` extends `IntentService`. We require few things, number one is that we do require a constructor and this constructor must call as super constructor with the name for the worker thread. So here it is, we have a constructor which is called as the super constructor.

(Refer Slide Time: 21:09)

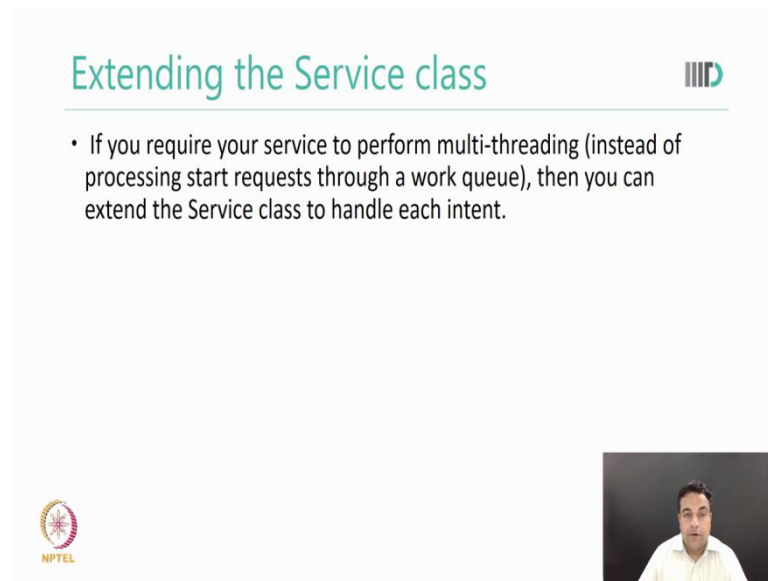
```
    /**  
     * A constructor is required, and must call the super IntentService\(String\)  
     * constructor with a name for the worker thread.  
     */  
    public HelloIntentService() {  
        super("HelloIntentService");  
    }  
  
    /**  
     * The IntentService calls this method from the default worker thread with  
     * the intent that started the service. When this method returns, IntentService  
     * stops the service, as appropriate.  
     */  
    @Override  
    protected void onHandleIntent(Intent intent) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            // Restore interrupt status.  
            Thread.currentThread().interrupt();  
        }  
    }  
}
```



Now the intent service calls this method from the default worker thread with the intent that started the service and when this method the `onHandleIntent` returns `IntentService` stops the

service as appropriate. So the second step that we have to do is to override the `onHandleIntent`, we do not do much we take the `onHandleIntent` and inside that actually speaking this is where you will be doing the actual work but because we are only demonstrating the concept we are just saying “Hey go and sleep for 5 seconds” and that is it, so that is the work that we are doing using the method.

(Refer Slide Time: 22:00)






The slide is titled "Extending the Service class" in a teal font. In the top right corner, there is a logo consisting of three vertical bars of increasing height. A horizontal line separates the title from the content. Below the line, there is a single bullet point: "• If you require your service to perform multi-threading (instead of processing start requests through a work queue), then you can extend the Service class to handle each intent." In the bottom left corner, there is a circular logo with a star-like pattern and the text "NPTEL" below it. In the bottom right corner, there is a small video inset showing a man with dark hair and a light-colored shirt speaking.

Now let us come to the second method of extending the service class instead of extending the intent. Now when we have such a simple method of using of extending intent service why would be like to extend the service class itself. So here is one reason giving to you, if you require your service to perform multi-threading, ok instead of passing start requests through a work queue, then you need to extend the service class to handle each intent.

(Refer Slide Time: 22:53)

## Example



```
public class HelloService extends Service {  
    private Looper mServiceLooper;  
    private ServiceHandler mServiceHandler;  
  
    // Handler that receives messages from the thread  
    private final s ServiceHandler extends Handler {  
        public ServiceHandler(Looper looper) {  
            super(looper);  
        }  
    }  
  
    @Override  
    public void handleMessage(Message msg) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
        try {
```



So let look at an example this is a rather long example so yes. Here we are trying to create a service by extending the base class service. I will only go through some of the major topics, so for example here we have to create a handler that receives a message from the thread, we will come to it later.

(Refer Slide Time: 23:04)

```
        super(looper);  
    }  
  
    @Override  
    public void handleMessage(Message msg) {  
        // Normally we would do some work here, like download a file.  
        // For our sample, we just sleep for 5 seconds.  
        try {  
            Thread.sleep(5000);  
        } catch (InterruptedException e) {  
            // Restore interrupt status.  
            Thread.currentThread().interrupt();  
        }  
  
        // Stop the service using the startId, so that we don't stop  
        // the service in the middle of handling another job  
        stopSelf(msg.arg1);  
    }  
}
```





So let us see our first override, our first override method is the handleMessage() method and here again just to demonstrate we are only download we are only sleeping for 5 seconds and then we stop the self-service but more importantly let us look at some of the other methods specially I am interested in the onCreate() method.



(Refer Slide Time: 23:32)

```
@Override
public void onCreate() {
    // Start up the thread running the service. Note that we
    create a
    // separate thread because the service normally runs in the
    process's
    // main thread, which we don't want to block. We also make
    it
    // background priority so CPU-intensive work will not disrupt
    our UI.

    HandlerThread thread = new
    HandlerThread("ServiceStartArguments",
        Process.THREAD_PRIORITY_BACKGROUND);
    thread.start();
```



So the onCreate() method is what starts up the thread running the service. So we are creating separate thread because the service normally runs in the main thread which we do not want to block and that is we do not want run the service in the main thread and we want to make it a backend thread so that we can even we can do CPU-intensive work. So it is still find we create a backend thread in the thread, thread type and then we start the thread. Then we get the HandlerThreads loop and use it for our handler. This is the same same loop and ServiceHandler that we had described earlier.



(Refer Slide Time: 24:44)

## EXAMPLE

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "service starting",
        Toast.LENGTH_SHORT).show();

    // For each start request, send a message to start a job and
    deliver the
    // start ID so we know which request we're stopping when
    we finish the job

    Message msg = mServiceHandler.obtainMess
    msg.arg1 = startId;
```



And then let us look at some other method that we have overridden. Another method is this onStartCommand() method, here we are just displaying a toast, so yes we do the toast and we



know which request we are stopping when we finish the job. And if we get killed we are returning something so first thing so here is the most important thing to look it is that what are we returning from onStartCommand() we return an int and this int has takes three values one of the value is START\_STICKY and very soon we will see that what other two values are and what is the meaning of this method.

(Refer Slide Time: 25:28)




```
@Override
public IBinder onBind(Intent intent) {
    // We don't provide binding, so
    return null;
}
```

The slide displays a code snippet for the `onBind()` method, which is annotated with `@Override`. The method returns `null`. A hand cursor is visible over the `return null;` line. The NPTEL logo is in the bottom left, and a video feed of the presenter is in the bottom right.

And then there is just a `onBind()` because we are just returning null and then there is `onDestroy()` where again we are just displaying toast. So let us go back to this method `onStartCommand()` in the int value that it is returning because that is the important bidding.

(Refer Slide Time: 25:50)



## Services

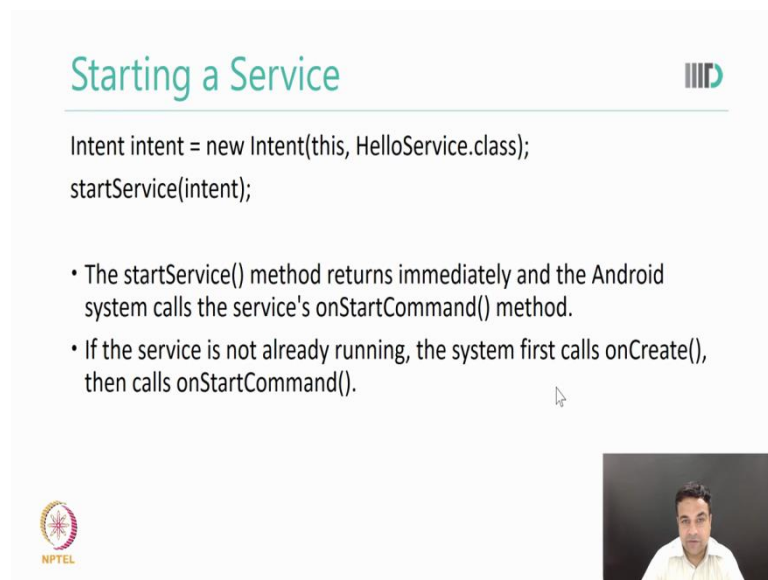
- `START_NOT_STICKY`
  - If the system kills the service after `onStartCommand()` returns, do not recreate the service, unless there are pending intents to deliver.
- `START_STICKY`
  - If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()`, but do not redeliver the last intent. Instead, the system calls `onStartCommand()` with a null intent, unless there were pending intents to start the service, in which case, those intents are delivered.
- `START_REDELIVER_INTENT`
  - If the system kills the service after `onStartCommand()` returns, recreate the service and call `onStartCommand()` with the last intent that was delivered to the service. Any pending intents are delivered in

The slide is titled 'Services' and lists the three return values of `onStartCommand()` with their respective behaviors. The NPTEL logo is in the bottom left, and a video feed of the presenter is in the bottom right.

So it can return one of the three things, one the first is `START_NON_STICKY` this means that if the system kills the service after `onStartCommand()` there is no need to recreate the service unless there are pending intents to deliver. The second is the `START_STICKY` and if the which means if the system kills the service after `onStartCommand()` we create the service and call `onStartCommand()` but do not redeliver the last intent instead the system calls on the `onSystemCommand()` with the null intent.

Then the third is the `START_REDELIVER_INTENT` in which if the system kills the service after `onStartCommand()` it returns recreate the service and call `onStartCommand()` with the last intent that is it sort of repeats everything. Now looking at the code you saw that how much easier it was by using by extending just intent service compare to by extending the service itself.

(Refer Slide Time: 27:10)



The slide is titled "Starting a Service" in a teal font. It contains a code snippet and two bullet points. In the bottom right corner, there is a small video feed of a man speaking. The NPTEL logo is in the bottom left corner.

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



- The `startService()` method returns immediately and the Android system calls the service's `onStartCommand()` method.
- If the service is not already running, the system first calls `onCreate()`, then calls `onStartCommand()`.

Now let us see that how do we start a service. So starting a service is same as starting an activity and it is very much the same thing. So you create an intent and from that intent then after you call the start service and then you pass the intent to it. So the start service method will return immediately and the Android system will call a service `onStartCommand()` method. Now if the service is not only running the system first calls `onCreate()` and then calls `onStartCommand()`.

(Refer Slide Time: 27:53)

## Stopping a service

- The system does not stop or destroy the service unless it must recover system memory and the service continues to run after `onStartCommand()` returns.
- The service must stop itself by calling `stopSelf()` or another component can stop it by calling `stopService()`.
- Once requested to stop with `stopSelf()` or `stopService()`, the system destroys the service as soon as possible.





Now once a service is started the system does not stop or destroy the service unless it must recover system memory and the service continues to run. I am emphasizing it again and again because this is very important point to remember that once you start a service, does not stop by itself and you must make the provisions so that the service stops either by calling `stopSelf()` from the service itself or another component calling stop by calling `stopService()`. So once requested to stop with `stopSelf()` or `stopService()`, the system destroys the service as soon as possible.

(Refer Slide Time: 28:43)

## Creating a Bound Service

- A bound service is one that allows application components to bind to it by calling `bindService()` in order to create a long-standing connection.
- You should create a bound service when you want to interact with the service from activities and other components in your application or to expose some of your application's functionality to other applications, through interprocess communication (IPC).
- To create a bound service, you must implement the `onBind()` callback method to return an `IBinder` that defines the interface for communication with the service.
- Other application components can then call `bindService()` the interface and begin calling methods on the service.

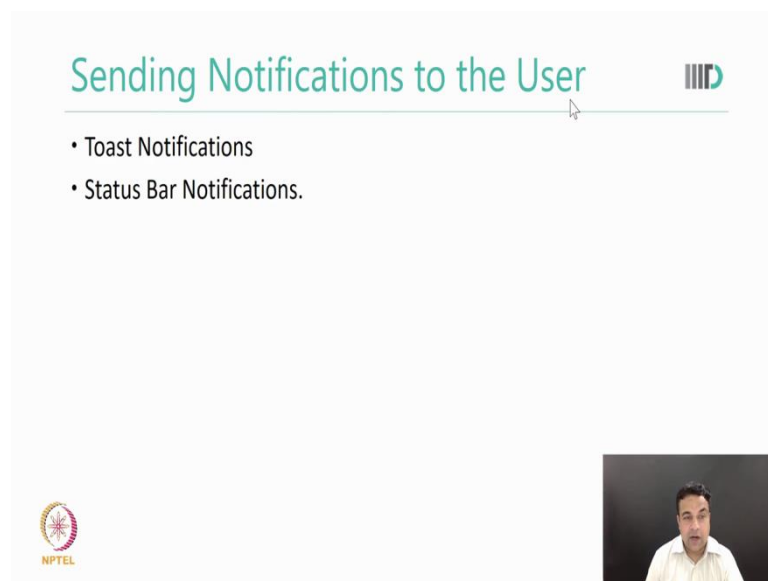


Now so far we were talking about creating a started service now let us also discuss a little bit of about creating a bound service. As we saw earlier bound service was mainly for IPC

communication and to bind other components to the service. Now a bound service is one that allows application component to bind to it by calling bind service in order to create a long-standing connection. And one should create a bound service only when you want to interact with the service from activities and other components in your application or to expose some of your application functionality to other application through IPC. Now in order to create a bound service you must implement the onBind() callback method, remember in other cases we return a null but here we will not return a null.

This method should return an IBinder that defines the interface of communication with the service, so what does it mean by interface communication with the service. As you know that this service needs to talk with other components. If they need to talk then there must be an interface over which we take talk and they can understand and this is the interface that we are talking about.

(Refer Slide Time: 30:03)



The slide is titled "Sending Notifications to the User" in a teal font. Below the title, there is a bulleted list: "• Toast Notifications" and "• Status Bar Notifications." In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.



- Toast Notifications
- Status Bar Notifications.

In other application component can then call bindService() to retrieve the interface and begin calling methods on the service. The bound service is pretty much like an IPC where you get the interface you find out how to interact with the service and then you start interacting with the service. Now service can send notification to the user in two ways, number one is a toast notification as we saw in our math quiz example and number two the status bar notification, so if you have been using Android phone you can see the status bar some of the messages which you see. So this is one of the method where which a service can talk to the user. So if there was a service which is for downloading a file after it has downloaded it can possibly set a status bar notification.

(Refer Slide Time: 30:45)

## Running a Service in the Foreground

- A foreground service is a service that's considered to be something the user is actively aware of and thus not a candidate for the system to kill when low on memory.
- A foreground service must provide a notification for the status bar, which is placed under the "Ongoing" heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.



Now, earlier we said that services run without the user interface in the background, you can actually run a service in the foreground as well. So a foreground service is a service that is considered to be something the user is actively aware of, for example I want to play the music so I would like the user to be actively aware of this. And because the foreground service is something a user is actively aware of, it is definitely therefore not a candidate for the system to kill when the system is low on memory. Now a foreground service must provide a notification for the status bar, which is placed under the Ongoing heading, which means that the notification cannot be dismissed unless the service is either stopped or removed from the foreground.

(Refer Slide Time: 31:36)

## Running a Service in the Foreground

```
Notification notification = new Notification(R.drawable.icon, getText(R.string.ticker_text),
    System.currentTimeMillis());

Intent notificationIntent = new Intent(this, ExampleActivity.class);
PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

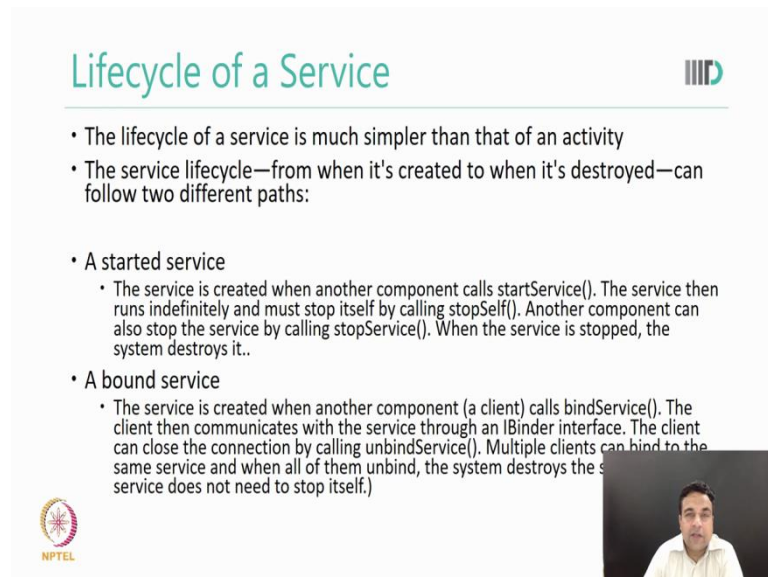
notification.setLatestEventInfo(this, getText(R.string.notification_title),
    getText(R.string.notification_message), pendingIntent);

startForeground(ONGOING_NOTIFICATION_ID, notification);
```



Let see, now let us see a program example of running a service in the foreground. So here is notification that must come and the most important thing to observe here is the this method `startForeground(ONGOING_NOTIFICATION_ID, notification)`. So notification is the way to indicate the user and start foreground is the way to start the services in program.

(Refer Slide Time: 32:18)



The slide is titled "Lifecycle of a Service" in a teal font. It features a list of bullet points explaining the service lifecycle. In the bottom right corner, there is a small video inset showing a man speaking. The NPTEL logo is in the bottom left corner.

- The lifecycle of a service is much simpler than that of an activity
- The service lifecycle—from when it's created to when it's destroyed—can follow two different paths:
- A started service
  - The service is created when another component calls `startService()`. The service then runs indefinitely and must stop itself by calling `stopSelf()`. Another component can also stop the service by calling `stopService()`. When the service is stopped, the system destroys it..
- A bound service
  - The service is created when another component (a client) calls `bindService()`. The client then communicates with the service through an `IBinder` interface. The client can close the connection by calling `unbindService()`. Multiple clients can bind to the same service and when all of them unbind, the system destroys the service. (The service does not need to stop itself.)

Now let us look at the lifecycle of a service, the lifecycle of a service is normally much simpler than that of an activity as we saw that activity lifecycle is depends very much on how user is interacting with the user interface that is provided by activity and because service does not provide a user interface the lifecycle of a service becomes much easier. So the service lifecycle that is when it is created to when it is destroyed can follow two different paths depending on the form of the service.

So if it is a started service then it is created when another component calls `startService()`. And then it runs indefinitely till either it calls `stopSelf()` or another component call `stopService()`. When the service is stopped, the system destroys it. On the other hand, the bound service is created when another component calls bind service and the client then communicates with the service through an `IBinder` interface, the client can close the connection by calling `unbindService()` and the multiple clients can bind to the same service when all of them unbind the service which is destroyed by the system itself.



(Refer Slide Time: 33:50)

# Implementing the life

```
public class ExampleService extends Service {  
    int mStartMode; // indicates how to behave if the service is killed  
    IBinder mBinder; // interface for clients that bind  
    boolean mAllowRebind; // indicates whether onRebind should be used
```

```
    @Override  
    public void onCreate() {  
        // The service is being created  
    }
```

```
    @Override  
    public int onStartCommand(Intent intent, int flags, int startId) {  
        // The service is starting, due to a call to startService()  
        return mStartMode;  
    }
```

```
    @Override  
    public IBinder onBind(Intent intent) {  
        // A client is binding to the service with bindService()
```



```
        return mStartMode;  
    }  
    @Override  
    public IBinder onBind(Intent intent) {  
        // A client is binding to the service with bindService()  
        return mBinder;  
    }
```

```
    @Override  
    public boolean onUnbind(Intent intent) {  
        // All clients have unbound with unbindService()  
        return mAllowRebind;  
    }
```

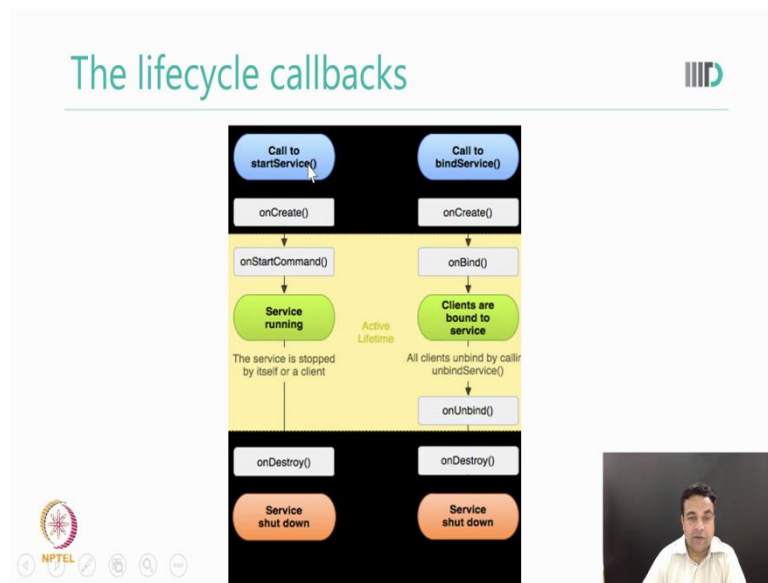
```
    @Override  
    public void onRebind(Intent intent) {  
        // A client is binding to the service with bindService(),  
        // after onUnbind() has already been called  
    }
```

```
    @Override  
    public void onDestroy() {  
        // The service is no longer used and is being destroyed  
    }  
}
```



Now let us look at the simple lifecycle callback methods. This is a very very simple example only giving you a very small idea of (the) how these methods look like there is onCreate(), onStartCommand(), onBind(), onUnbind(), onRebind(), onDestroy() and as we did in the activity it has the good practice that when you write your first service program just override them put a long message here, so that we know when these methods are being called.

(Refer Slide Time: 34:23)



Here is a graphically representation of these methods as you see that call to `startService()` comes `onCreate()` comes, so here is when the service is running that is after the `onStartCommand()` and when the service is stopped by itself or a client for `onBind()` again after the `onBind()` command and when the last `onUnbind()` is being called, then service call `onDestroy()`, `onDestroy()` is calling by services shut down. So this was all about services which is one of the most important app component besides activities I have given you small program `()` (34:58) in this video. I would soon be uploading a program running a service, but before that you can actually use this `()` (35:06) and create your own program where you create a service and maybe do not do much but just override on the callback methods and put a long messages and then just try to see the lifecycle of a service, thank you.