ESCUELA DE SISTEMAS Y TECNOLOGÍAS

Transparencias de **ANALISTA DE SISTEMAS**

Edición 2020 - Materia: **Aplicaciones Android**

TEMA: Servicios

Consideraciones

- Estas transparencias **no** tienen el objetivo de suplir las clases.
- Por tanto, serán **complementadas** con ejemplos, códigos, profundizaciones y comentarios por parte del docente.
- El **orden** de dictado de estos temas está sujeto a la consideración del docente.

Referencias

- Documentación para developers:
  - *http://developer.android.com/*
- Servicios:
  - *https://developer.android.com/guide/components/services.html*
- Servicios enlazados:
  - *https://developer.android.com/guide/components/bound-services.html*
- Procesos y subprocesos:
  - *https://developer.android.com/guide/components/processes-and-threads.html*
- Programando alarmas recurrentes:
  - *https://developer.android.com/training/scheduling/alarms.html*

Agenda

- Servicios
- Enlazar Servicios a Actividades
- Creación de Servicios en Foreground
- Tareas Asincrónicas
- IntentServices
- Threads
- Alarmas

Servicios (1)

- A diferencia de las Actividades, que muestran las interfaces gráficas, los Servicios corren invisiblemente.
- Si la aplicación proporciona una funcionalidad que no depende directamente de la entrada del usuario, o incluye operaciones que consumen mucho tiempo, debe utilizarse un Servicio.
- Si el Servicio es percibido directamente por el usuario (por ejemplo, mediante la reproducción de música), puede ser necesario aumentar su prioridad mediante el etiquetado como un componente de primer plano.
- Para **definir** un Servicio, se debe crear una nueva clase que extienda de **Service**. Se deben redefinir los métodos **onCreate** y **onBind**.

Servicios (2)

- Después de haber construido un nuevo servicio, se debe registrar en el manifiesto de la aplicación. Para ello, se incluye una etiqueta de servicio en el nodo de la aplicación:

  <service android:enabled="true" android:name=".MyService"/>
- Luego para asegurarse que el Servicio puede iniciarse y detenerse solamente por la propia aplicación, debe agregarse un atributo **permission** para el nodo de servicio.
- Para ejecutar un servicio se debe reemplazar el controlador de eventos **onStartCommand** para ejecutar la tarea (o comenzar la operación en curso) encapsulado por el Servicio. También se puede especificar el comportamiento de reinicio del servicio (en caso de que el sistema lo destruya) dentro de este controlador.

Servicios (3)

- El método **onStartCommand** se llama cada vez que se inicia el servicio utilizando ***StartService***, por lo que puede ser ejecutado varias veces durante la vida de un servicio.
- El valor de retorno controla el comportamiento de reinicio mediante la devolución de una de las siguientes constantes de servicio:
  - **START_STICKY**: Si el sistema cierra el servicio después de que se devuelva *onStartCommand*, vuelve a crear el servicio y llama al *onStartCommand*, pero no vuelve a entregar la última ***intent***, sino una ***intent*** nula, a menos que existan ***intents*** pendientes para iniciar el servicio, en cuyo caso se entregarán estas últimas.

Servicios (4)

- **START_NOT_STICKY**: Si el sistema cierra el servicio después de que se devuelve ***onStartCommand***, no vuelve a crear el servicio, salvo que existan ***intents*** pendientes para entregar.
- ***START_REDELIVER_INTENT***: Si el sistema cierra el servicio después de que se devuelve ***onStartCommand***, vuelve a crear el servicio y llama a ***onStartCommand*** con la última ***intent*** que se entregó al servicio. A su vez, se entregan todas las ***intents*** pendientes.

Servicios (5)

- Usando el Parámetro **flag** de ***onStartCommand*** se puede determinar de qué manera ha sido iniciado el servicio:
  - **START_FLAG_REDELIVERY**: Indica que el reinicio es causado por el sistema después de haber terminado el servicio antes de que fuera detenido explícitamente por una llamada a stopSelf.
  - **START_FLAG_RETRY**: Indica que el servicio se ha reiniciado después de una terminación anormal. Esto sucede cuando el servicio se estableció previamente START_STICKY.

Servicios (6)

- Para iniciar un servicio, se debe llamar a **StartService**. Se puede hacer de dos maneras: Implícitamente o Explícitamente, y si el servicio requiere permisos que la aplicación no tiene, la llamada a **StartService** lanzará una ***SecurityException***.
- Para detener un servicio, se debe usar **stopService**. Se puede hacer de dos maneras: Implícitamente o Explícitamente, especificando el servicio requerido.

Enlazar servicios a Actividades (1)

- El enlace es útil para las actividades que se beneficiarían de una interfaz más detallada con un Servicio. Para ello se requiere implementar el método **onBind**, que devuelve la instancia actual del Servicio enlazado.
- La conexión entre un Servicio y otro componente se representa con una **ServiceConnection**. Para enlazar un servicio a otro componente de la aplicación, es necesario implementar un nuevo **ServiceConnection**, redefiniendo los métodos ***onServiceConnected*** y ***onServiceDisconnected*** para obtener una referencia a la instancia de Servicio después de que una conexión se haya establecido.

Enlazar servicios a Actividades (2)

- Para realizar la unión, se debe llamar al **bindService** dentro de la Actividad, que seleccione el Servicio al cual se enlazará, y una instancia de ***ServiceConnection***. También se puede especificar banderas de enlace.
- **Nota**: Una vez que el Servicio ha sido enlazado, todos sus métodos y propiedades públicas están disponibles a través del objeto ***IBinder*** obtenido del manejador ***onServiceConnected***.

Creación de servicios en Foreground (1)

- Cuando se quiere calcular qué aplicaciones y qué componentes de aplicaciones son destruidos, Android tiene en cuenta la prioridad (los Servicios están por debajo de las Actividades).
- En los casos en que el Servicio esté siendo percibido directamente por el usuario, puede ser apropiado subir su prioridad para que equivalga al de una Actividad de primer plano. Esto puede realizarse a través del llamado al método **startForeground**.
- Debido a que se espera que los Servicios de primer plano tengan notoriedad directa para el usuario (por ejemplo, mediante la reproducción de música), las llamadas a ***startForeground*** deben especificar una notificación de que éste se encuentra en curso.

Creación de servicios en Foreground (2)

- Es una buena práctica proporcionar una manera simple a los usuarios que les permita desactivar un servicio de primer plano - típicamente se hace desde cualquier actividad que se encuentre abierta haciendo clic en la notificación en curso (o desde la notificación en sí misma).
- Cuando un Servicio ya no requiere prioridad en primer plano, se puede mover de nuevo a un segundo plano, y quitar opcionalmente la notificación en curso utilizando el método **stopForeground**. La notificación se cancelará automáticamente si el servicio se detiene o se termina.

Tareas Asincrónicas (1)

- La clase **AsyncTask** implementa un modelo de mejores prácticas para mover las operaciones que consumen mucho tiempo a un Hilo de background y sincronizarla con el hilo de interfaz de usuario para las actualizaciones y cuando el tratamiento se ha completado. Ofrece la comodidad de los controladores de eventos sincronizados con el hilo GUI que le permite actualizar Vistas y otros elementos de interfaz de usuario para informar el progreso o publicar resultados cuando la tarea se ha completado.
- **AsyncTask** maneja toda la creación del hilo, la gestión, y la sincronización, lo que le permite crear una tarea asincrónica que consiste en el procesamiento que va a ser realizado en el background y actualizaciones de interfaz de usuario donde ambas tienen que ejecutarse durante el procesamiento.

Tareas Asincrónicas (2)

- Para crear una nueva tarea asincrónica, se debe extender la clase **AsyncTask**, especificando los tipos de parámetros a usar, o colocando ***Void*** para los que no quiera especificar.
- Después de haber implementado una tarea asincrónica, se debe ejecutar mediante la creación de una nueva instancia y llamando al método **execute()**.
- Cada instancia AsyncTask puede ejecutarse sólo una vez. Si intenta llamar a ejecutar una segunda vez, se produce una excepción.

Tareas Asincrónicas (3)

- La clase **Loader** abstracta se introdujo en Android 3.0 (nivel de API 11) para encapsular la mejor práctica para la carga de datos asincrónica dentro de los elementos de interfaz de usuario, tales como Actividades y fragmentos.
- Crear su propia implementación de Loader, suele ser una mejor práctica, que extender la clase **AsyncTaskLoader** ,en lugar de la clase Loader directamente. En general los **Loaders** personalizados deben:
  - Cargar datos de forma asincrónica.
  - Monitorear la fuente de los datos cargados y automáticamente proporcionar resultados actualizados

IntentServices

- Un **IntentService** es un contenedor que implementa la mejor práctica de Servicios de ***Background*** para realizar tareas **on demand**, como actualizaciones de Internet o de procesamiento de datos.
- Para implementar un servicio como un ***IntentService,*** se debe redefinir el manejador **onHandleIntent**
- El onHandleIntent se ejecutará en un hilo de trabajo una vez por cada Intent

Thread (1)

- Aunque el uso de Servicios **Intent** y la creación de **AsyncTasks** son atajos útiles, hay momentos en los que se desea crear y administrar sus propios hilos para realizar el proceso de background. Esto suele suceder cuando se tiene una larga ejecución o hilos interrelacionados entre sí que requieren una gestión más sutil o compleja que la proporcionada por las dos técnicas descritas hasta ahora.
- Se pueden crear y gestionar Hilos Hijo usando la clase **Handler** de Android y las clases heredadas disponibles dentro de **java.lang.Thread**
- Si se está ejecutando dentro de una Actividad, también se puede utilizar el método **runOnUiThread**, que permite forzar un método para ejecutar en el mismo Hilo que la Actividad de interfaz de usuario.

Thread (1)

- Hilos Background
- La Responsividad es uno de los atributos más importantes de una buena aplicación Android. Para asegurar que su aplicación responde rápidamente a cualquier interacción del usuario o de eventos del sistema, es vital que se muevan todos los procesamientos y operaciones de E / S del Hilo Principal de la aplicación a un Hilo Hijo.
- Es importante utilizar Hilos de background para cualquier proceso no trivial que no interactúe directamente con la interfaz de usuario. Es particularmente importante para programar operaciones de archivos, operaciones de búsqueda de la red, transacciones de bases de datos y cálculos complejos, en un subproceso en segundo plano.

Alarmas (1)

- Las alarmas son un medio para enviar Intents en momentos o intervalos predeterminados. A diferencia de los temporizadores, las alarmas operan fuera del ámbito de su aplicación, por lo que pueden ser utilizadas para desencadenar eventos o acciones, incluso después de que la aplicación ha sido cerrada.
- Las alarmas son especialmente potentes cuando se utilizan en combinación con Receptores Broadcast, lo que le permite configurar alarmas para iniciar los servicios, o incluso abrir actividades, sin que la aplicación esté abierta o corriendo.

Alarmas (2)

- Las Alarmas en Android permanecen activas mientras el dispositivo está en *Sleep Mode* y, opcionalmente, se puede configurar para "despertar" el dispositivo; sin embargo, todas las alarmas se cancelan cada vez que se reinicia el dispositivo.
- Las operaciones de alarma se manejan a través del ***AlarmManager***, un servicio del sistema al cual se accede a través ***getSystemService***
- Para crear una Alarma **one-shot**, se usa el método **set**, se especifica el tipo de alarma, el tiempo de **trigger**, y un **PendingIntent** para activar una vez que el **trigger** se complete. Si el tiempo de **Trigger** especificado está en el pasado, entonces la alarma comenzara a activarse inmediatamente.

Alarmas (3)

- Los tipos de alarmas disponibles son los siguientes:
  - **RTC_WAKEUP** : Activa el dispositivo del "sleep mode" para disparar el PendingIntent a la hora especificada.
  - **RTC**: Activa el PendingIntent especificado pero no despierta el dispositivo.
  - **ELAPSED_REALTIME**: Activa el PendingIntent en base a la cantidad de tiempo transcurrido desde que el dispositivo se inicia pero no despierta el dispositivo. El tiempo transcurrido incluye cualquier período de tiempo que el dispositivo permanecía dormido.
  - **ELAPSED_REALTIME_WAKEUP**: Despierta el dispositivo y dispara el PendingIntent después de transcurrido un período determinado de tiempo desde el arranque del dispositivo.

Alarmas (4)

- Para cancelar una alarma, se debe llamar a **cancel** en el *Alarm Manager*, especificando que ya no se debe disparar mas el trigger
- Para configurar alarmas repetitivas, se usan los mismos criterios que los usados para las **one-shot**, pero en el trigger de intervalo se especifica la cantidad.
- Se usan los métodos **setRepeating** (cuando se necesita un control detallado sobre el intervalo exacto de la alarma a repetir. El valor de intervalo pasado a este método le permite especificar un intervalo exacto para la alarma, hasta el milisegundo) o **setInexactRepeating** (ayuda a reducir el consumo de la batería asociada a despertar el dispositivo en un horario regular para realizar actualizaciones) del **AlarmManager**