**Apuntes desarrollo en dispositivos móviles**

**Variables con tipología importante:**

**Concepto de variable atómica:** Son variables que implementan automáticamente opciones de sincronizado y que garantizan que ciertas operaciones fundamentales sobre ellas van a ser realizadas en forma de hebra segura. Las operaciones atómicas son operaciones que no deben ser realizadas por dos hilos de manera simultánea .

**Variable ThreadLocal:** Se trata de variables que son locales a las diferentes clases sobre las que se ejecuta una hebra y es común a ellas, es decir, mantendrá una instancia de esa variable hebra por hebra.

**Variable volátil:** Se refiere, en pocas palabras, a una variable cuyo valor va a ser modificado por diferentes hebras. Si una variable es volátil se espera que vaya a ser modificada desde diferentes hebras.

**Ciclo de trabajo del desarrollo en Android:**

En programación con Android funcionamos principalmente con la clase Activity y por tanto tenemos que conocer el ciclo de vida de la actividad propiamente dicha.

La actividad puede estar en tres estados diferentes:

Corriendo, pausada o Parada.

El estado de pausa puede ser matado por el sistema de Android completamente, pasa lo mismo cuando la aplicación está parada.

Una actividad en pausa o parada puede recuperarse del todo pues las mismas instancias están todavía en memoria.

Una aplicación goza de una serie de métodos protegidos que dictaminan el estado en el que se encuentra la aplicación en cada momento:

**onCreate():**

Se llama cuando nuestra actividad empieza por primera vez. Aquí se setearán todos los elementos del UI. Se llama una única vez en el ciclo de vida de nuestra actividad.

**onRestart():**

Se llama cuando la actividad prosigue después de un estado de detención. Viene precedida, de esta manera, por una función onStop()

**onStart():**

Se llama inmediatamente después de onCreate() o cuando la actividad se retoma o prosigue desde un estado de parada, en este caso se ve precedida por el método onRestart()

**onResume()**

Se llama inmediatamente después de la función onStart() o cuando la actividad se retoma desde un estado de pausa (como cuando se desbloquea la pantalla).

**onPause()**

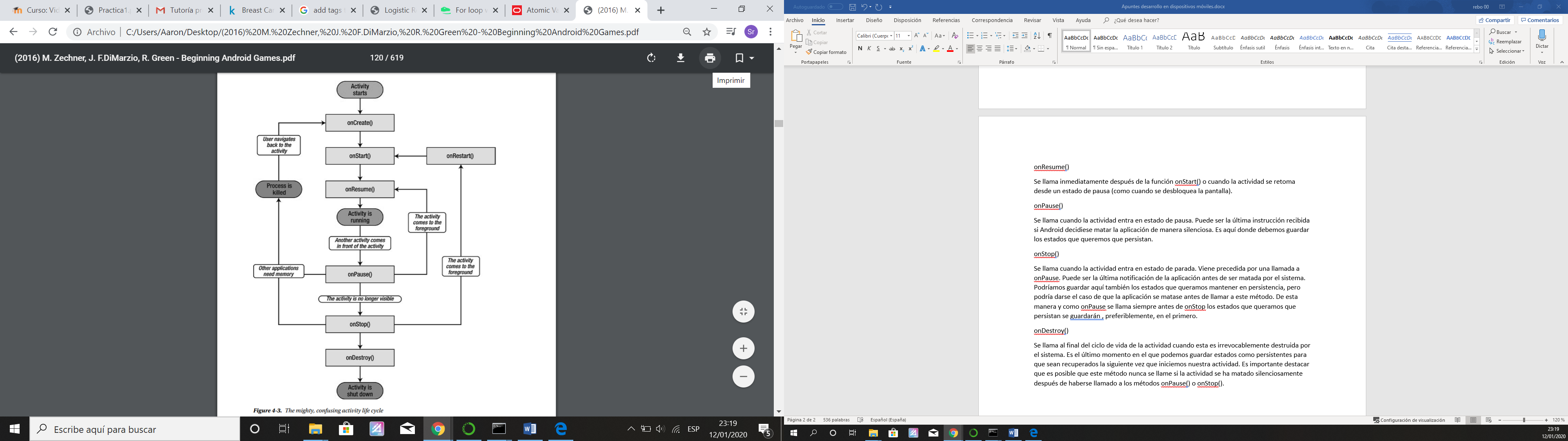
Se llama cuando la actividad entra en estado de pausa. Puede ser la última instrucción recibida si Android decidiese matar la aplicación de manera silenciosa. Es aquí donde debemos guardar los estados que queremos que persistan.

**onStop()**

Se llama cuando la actividad entra en estado de parada. Viene precedida por una llamada a onPause. Puede ser la última notificación de la aplicación antes de ser matada por el sistema. Podríamos guardar aquí también los estados que queramos mantener en persistencia, pero podría darse el caso de que la aplicación se matase antes de llamar a este método. De esta manera y como onPause se llama siempre antes de onStop los estados que queramos que persistan se guardarán, preferiblemente, en el primero.

**onDestroy()**

Se llama al final del ciclo de vida de la actividad cuando esta es irrevocablemente destruida por el sistema. Es el último momento en el que podemos guardar estados como persistentes para que sean recuperados la siguiente vez que iniciemos nuestra actividad. Es importante destacar que es posible que este método nunca se llame si la actividad se ha matado silenciosamente después de haberse llamado a los métodos onPause() o onStop().



En resumidas cuentas:

Antes de entrar en el estado de ejecución (running) el método onResume() siempre se ejecuta de manera previa, vengamos o no de un estado de parada o pausa.

La actividad puede ser destruida de manera silenciosa después de ejecutarse onPause. No debemos asumir nunca que los métodos de onPause y onStop se van a llamar. Será en onPause donde deberemos de asegurarnos de guardar todos los estados que queramos que persistan.

Cabe la posibilidad de que onDestroy nunca se llame si el sistema decide matar la aplicación después de onPause o onStop. Sin embargo, es interesante conocer a veces cuando la aplicación va a ser realmente destruida. De esta manera aparece el método isFinishing que podemos llamar en cualquier momento para verificar si la aplicación va a ser efectivamente eliminada o matada. Lo único que tenemos que hacer es por tanto llamar a este método en onPause para asegurarnos si efectivamente nuestro sistema va a matar la aplicación o no.

De esta manera nos tenemos que quedar con que los métodos con los que vamos a jugar para gestionar nuestra aplicación van a ser:

onCreate: Inicializaremos la ventana y los componentes del UI y de donde recibiremos el input.

onResume(): Empezaremos o relanzaremos nuestra hebra de loop principal.

onPause(): Detenemos o pausamos nuestra hebra principal y si la actividad isFinishing devolvemos verdadero. Aprovechamos para guardar toda la información y estados que queramos que persistan.

**Chascarrillos de Java:**

***Concepto de interfaz:***

Una interfaz es una clase abstracta que contiene declaraciones de métodos (no implementados) y que no posee atributos.

“Es una colección de métodos abstractos y propiedades constantes. Son las clases que implementan las interfaces las que describen la lógica del comportamiento de los métodos que se declaran en dichas interfaces.

No son clases, solo especifican requerimientos para las clases que las implementan

Son una forma de especificar qué debe hacer una clase, pero sin especificar cómo”

Las interfaces son las que nos permiten contar en Java con herencia múltiple, pues no existe herencia múltiple de Clases en Java. Trabajando con clases solo existe herencia simple.

***Clase abstracta***

*Son clases abstractas aquellas clases de las que no se permite la creación de objetos.*

***Métodos abstractos:***

No tienen cuerpo, solo existen dentro de una clase abstracta y deben de estar sobre escritos o reimplementados en las subclases. Si una subclase no redefinir todos los métodos de la clase padre es una subclase abstracta. Para que la subclase sea concreta es necesario redefinir todos los métodos de la superclase abstracta.

Un método abstracto es un método que nunca va a ser ejecutado porque no tiene cuerpo, no pueden ser implementados en la clase base.

***El problema del diamante:***

Este se da cuando se heredan de clases que a su vez heredan de otra común. En Java, C++ y C# es imposible evitarla pues todas las clases heredan necesariamente de Object.

En el caso de Java y las interfaces la herencia múltiple no acarrea problemas como este puesto que no se implementan métodos en estas ni se declaran atributos, esto es, no hay sobrecarga de operadores.

Por defecto se hereda siempre de la clase Object.

No existe herencia múltiple.

“Final” deshabilita la vinculación dinámica y permite que una variable quede asociada a un valor único durante todo el proceso, de manera inmutable. Solo se le puede asignar un valor o un objeto una única vez y una vez definido el valor no se podrá modificar. En caso de querer hacerlo el compilador nos dará un error.

Los strings en Java son inmutables.

Si una clase tiene métodos abstractos a de ser a la fuerza una clase abstracta.

Las excepciones nos permiten capturar errores que saltan en el hilo de ejecución.