**Ciclo de vida**

* **Create():** se llama cuando la aplicación se crea por primera vez
* **Resize():** Se llama cuando al aplicación se redimensiona. Puede ocurrir en cualquier momento en el que la aplicación este corriendo pero nunca ocurrirá antes de llamar al método create(). El evento solo es posible en la aplicación de escritorio
* **Pause()**: Se llama cuando la aplicación se pausa. Una aplicación es pausada antes de que sea destruida, cuando un usuario presionó el botón Home de Android o ha recibido una llamada. En escritorio solo será llamado inmediatamente antes de que el método dispose() sea llamado.
* **Dispose()**: Llamado cuando la aplicación es destruida. Precedido por una llamada a pause().
* **Render()**: Se llama cuando la pantalla debe renderizarse a sí misma.
* **Resume()**: Se llama cuando la aplicación vuelve del estado pause(). En Android ocurre cuando la aplicación obtiene el foco de nuevo. En escritorio este método nunca será llamado.

**Application, StarterClass y ApplicationListener**

Application es el proyecto core, o sea el punto de entrada principal del proyecto. Setea una ventana y una superficie de renderización y maneja los distintos aspectos de la aplicación: Gráficos, audio, inputs y archivos.

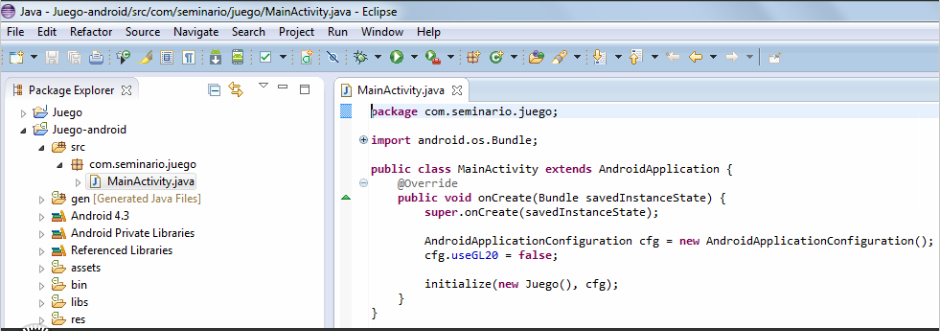
Al haber creado los proyectos para Android, Desktop, iOS y HTML5, tendremos varias implementaciones de esta interfaz:

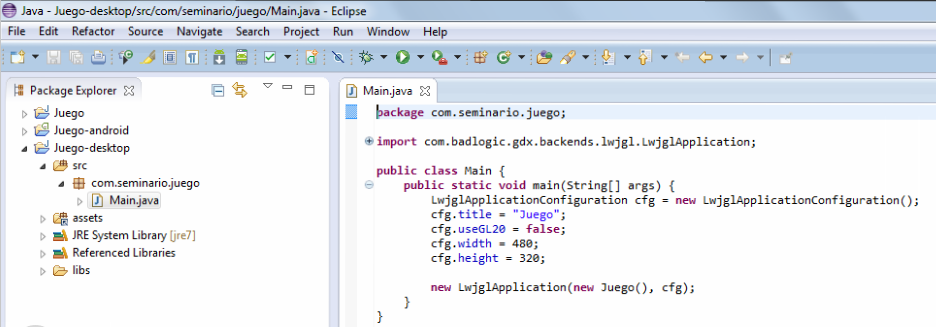
* AndroidApplication (Android)
* LwjglApplication (Escritorio)
* IOSApplication.Delegate (iOS)
* GwtApplication (HTML5)

La StarterClass es el único código que necesita ser escrito para cada plataforma específica es el de las "starter classes". Para cada plataforma, una porción de código dentro de estas clases instanciará una implementación concreta de la interfaz "Application".

Al crear el proyecto con la GUI que nos brinda LibGDX, estas clases serán creadas automáticamente con código por defecto que puede ser modificado.

Ejemplo de proyectos para Android y Desktop:





En el método inicializador de cada starter class, se especifica una configuración, y se crea luego una instancia de la aplicación. Para ello se pasan dos parámetros

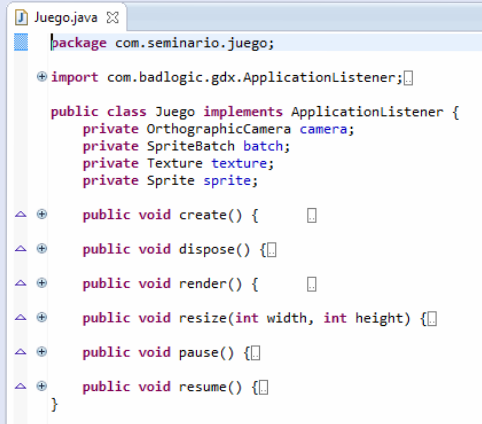
1. Una nueva instancia de la implementación de ApplicationListener que se encuentra en el proyecto del core. (En el proyecto se llama Juego.java).
2. La configuración especificada.

La aplicación luego llamará a los métodos del "ApplicationListener" en los momentos apropiados.

En la implementación de la interfaz ApplicacionListener se le da comportamiento a los eventos del ciclo de vida (create(), resize(), render(), pause(), dispose() y resume()), y se le pasa una instancia de esa implementación a la Aplicación de cada una de las plataformas para las que se desee desarrollar el juego.

La aplicación la llamará cada vez que un evento ocurra.

En esta imagen se puede ver la implementación de ApplicationListener que se crea por defecto al crear un proyecto:



LibGDX está compuesto por cuatro módulos, a los cuales se accede por medio de la clase estática Gdx:

• **Input**: Gestiona la entrada a través de teclado, pantalla táctil, acelerómetro, etc.

• **Graphics**: Permite gestionar la representación de imágenes en la pantalla.

• **Files**: Se trata de un componente para lectura y escritura de ficheros de datos como imágenes, archivos de configuración, sonidos, música, texturas, etc.

• **Audio**: Facilita la reproducción y grabación de audio en todas las plataformas que soporta.

**Input**

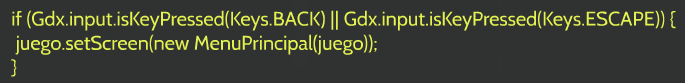
En desktop, los usuarios se pueden comunicar con la aplicación por medio del tecladoy el mouse. En Android, el mouse es reemplazado por una pantalla táctil, y puede existir un teclado físico. Todos los dispositivos compatibles con Android también poseen acelerómetroy algunos un compass(sensor de campo magnético).

LibGDX abstrae todos esos dispositivos de entrada. El mouse y las pantallas táctiles son tratadas como si fueran lo mismo.

Para la entrada de datos por parte del usuario se puede hacer polling o registrar un listener que recibirá eventos de entrada en orden cronológico.

Todas las facilidades de entrada son accedidas por medio del módulo Input de LibGDX.

Ejemplo:



**Graphics**

El módulo Graphics provee información acerca del display del dispositivo actual y la ventana de la aplicación como información y acceso al contexto de OpenGL.

Como el resto de los módulos, se accede a través de los campos estáticos de la clase Gdx.

Uno de los métodos más útiles en la clase Graphics es getDeltaTime(), el cual provee el tiempo transcurrido desde el último cuadro renderizado. Esto puede ser útil para animaciones basadas en tiempo.

Otro método útil es getFramesPerSecond(), que retorna un promedio del rango de cuadros para propósitos de diagnóstico.

**Texture, TextureRegion y SpriteBatch**

**Texture**

Una imagen que es subida al GPU (unidad de procesamiento de graficos) es denominada una textura (Texture). Las medidas de las imágenes deben ser potencias de dos, salvo que se trate de una aplicación que utiliza OpenGL 2.0.



**TextureRegion**

Una forma que representa una porción de una textura es denominada una región de textura (TextureRegion):



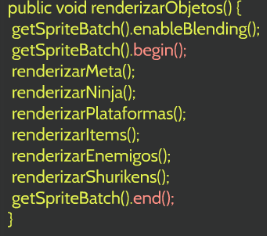
En el ejemplo anterior, se crea una región de textura de 32 x 32 píxeles, partiendo desde el punto 0,0 de la textura.

**SpriteBatch**

Es común dibujar una textura mapeada a un rectángulo, y dibujar la misma textura o una región de la textura varias veces. Sería ineficiente mandar cada rectángulo al GPU para que sea dibujado. Por eso, muchos rectángulos para la misma textura pueden ser descriptos y mandados al GPU al mismo tiempo por medio de la clase SpriteBatch.

A este se le manda una textura y las coordenadas de cada rectángulo. Este recolecta la forma geométrica sin enviársela al GPU.

Todas las llamadas para dibujar con SpriteBatch deben ser realizadas entre sus métodos begin() y end(). Esto se puede ver en el siguiente fragmento de código del juego:



Para dibujar texturas utilizando un SpriteBatch, se utiliza su método draw(). Posee varias sobrecargas para dibujar texturas y regiones de texturas.

**Files**

El modulo Files de libGDX permite:

* Leer un archivo.
* Escribir en un archivo.
* Copiar un archivo.
* Mover un archivo.
* Eliminar un archivo.
* Listar archivos y directorios.
* Averiguar si un archivo o directorio existe.

Cada una de las plataformas maneja el I/O de archivos de una forma diferente, pero gracias a este módulo el programador no tiene que preocuparse por ello.

Tipos de archivos

Un archivo en libGDX es representado por una instancia de la clase FileHandle. Esta posee un tipo, que define en donde se encuentra el archivo:

* Classpath: Almacenados en las carpetas fuentes. Son empaquetados con los jars y siempre son de solo lectura. Deben ser evitados si es posible.
* Internal: Se encuentran en la carpeta assets del proyecto de Android que está linkeada al resto de los proyectos. Sólo se pueden leer archivos.
* Local: Almacenamiento interno. En escritorio los archivos están juntos al JAR o en la carpeta del proyecto. En Android están en el almacenamiento interno del teléfono.
* External: Relativos a la raíz de la tarjeta SD en Android, y en la carpeta personal del usuario en desktop. Se recomienda su uso para datos volátiles, es decir, que no son imprescindibles para la aplicación.
* Absolute: Necesitan tener sus rutas completas especificadas. Por temas de portabilidad esta opción debe ser solo utilizada cuando sea absolutamente necesaria.

Existen cinco métodos en Gdx.files:

• Gdx.files.classpath(String): FileHandle

• Gdx.files.internal(String): FileHandle

• Gdx.files.local(String): FileHandle

• Gdx.files.external(String): FileHandle

• Gdx.files.absolute(String): FileHandle

(String: ruta del archivo)

Se pueden tener Fileandles de archivos que no existan. Si se intenta escribir sobre ellos, se crearán automáticamente. Si se intenta leer sobre ellos dará un error.

Por ejemplo, para crear una textura a partir de un archivo que se encuentra en la carpeta assets (internal):

**Texture texturaFondo = new Texture(Gdx.files.internal("fondo\_menu.png"));**

En este otro ejemplo del juego se obtiene un FileHandle que representa un archivo externo donde están guardados los datos del juego:

**FileHandle fh = Gdx.files.external("ninjump.json");**

Algunos métodos de lectura:

* readString()
* readBytes()

Algunos métodos de escritura:

* writeString(String, boolean);
* writeBytes(byte[], boolean);

**Audio**

LibGDX provee métodos para reproducir efectos de sonidos cortos, o música desde el disco. También provee acceso al hardware de audio. El acceso a las facilidades de audio que brinda se hace por medio del módulo audio.

LibGDX pausa el audio cuando la aplicación está pausada y vuelve a reproducirlo cuando se regresa a la aplicación.

**Efectos de sonido**

Los efectos de sonido son pequeños archivos de audio de unos pocos segundos, LibGDX soporta formatos MP3, OGG y WAV. Estos están representados por la interfaz Sound. Para cargar un efecto de sonido:

Sound sonido = Gdx.audio.newSound(Gdx.files.internal("miSonido.mp3"));

El código anterior carga un archivo de audio llamado “miSonido.mp3” de la carpeta assets.

El siguiente método reproduce el sonido una vez, con el volumen al máximo. Retorna un long con el id de la instancia de sonido que se está reproduciendo.

**long id = sonido.play(1.0f);**

El método stop detiene el sonido, recibiendo como parámetro el id de la instancia.

**sonido.stop(id);**

Sound posee métodos para el procesamiento del audio, como por ejemplo:

* setPitch(): cambia el tono del sonido.
* setPan(): cambia el paneo del sonido, siendo -1 la izquierda y 1 la derecha.
* setLooping(): hace que el sonido se reproduzca como un bucle.

Una vez que no se necesite, hay que asegurarse de desecharlo, y de no acceder a él una vez desechado ya que provocará errores:

**sonido.dispose();**

**Música**

Para cualquier sonido que sea más largo que varios segundos es preferible reproducirlo desde el disco que cargarlo completamente en la RAM. Para ello LibGDX provee la interfaz Music.

Para cargar una instancia de Music:

**Music musica = Gdx.audio.newMusic(Gdx.files.internal("musica.mp3"));**

Para reproducir la música se utiliza el método play():

**musica.play();**

También posee otros métodos para el procesamiento de la música al igual que Sound:

*musica.setVolume(0.5f);*

*musica.setLooping(true);*

*musica.stop();*

*musica.pause();*

*musica.play();*

*boolean isPlaying = musica.isPlaying();*

*boolean isLooping = musica.isLooping();*

*float position = musica.getPosition();*

Las instancias de Music son pesadas, por lo que se deben desechar si dejan de ser necesitadas para liberar recursos:

**musica.dispose();**

**Scene2D**

Scene2D se utiliza para crear aplicaciones e interfaces de usuario utilizando una jerarquía de actores. Provee las siguientes funcionalidades:

* La rotación y la escala de un grupo son aplicadas a todos los actores hijos.
* Dibujo simplificado a través de SpriteBatch.
* Detección de colisiones de actores. Cada uno determina si colisionó.
* Ruteo de eventos de entrada u otros eventos al actor apropiado.
* Sistema de acciones para la manipulación fácil de actores en el tiempo.

Scene2D está bien equipado para crear menús de juegos, HUDS, herramientas, y otras interfaces de usuario. El paquete scene2d.ui provee varios actores y otras utilidades específicamente para la creación de interfaces de usuario.

Los actores almacenan información que es considerada parte del modelo pero también son la vista, ya que saben cómo dibujarse. Esto hace que la separación MVC sea difícil. Cuando es utilizado sólo para interfaces de usuario o aplicaciones que no utilizan MVC, esto no es un problema.

Scene2D posee tres clases en su núcleo.

* La clase Actor, con una posición, medida rectangular, punto de origen, escala, rotación y color.
* La clase Group, que es un actor que puede tener otros actores hijos.
* La clase Stage posee una cámara, un SpriteBatch, y un grupo. Administra el dibujado de los actores y distribuye los eventos de entrada.

Cuando se llama al método draw del Stage, este llama al draw de todos sus actores. Lo mismo sucede cuando se llama a su método act.

El método draw de un actor dibuja una región utilizando la información del actor. El SpriteBatch que se le pasa para dibujar es configurado para dibujar en las coordenadas del padre, por lo que 0,0 es el borde inferior izquierdo del padre. Esto hace que dibujar sea simple, por más que el padre esté rotado y escalado.

**InputListener**

Para la mayoría de tipos de eventos, se utilizan listeners específicos por conveniencia. Por ejemplo, InputListener se utiliza para recibir y manejar eventos de entrada. Un actor sólo necesita agregar un InputListener para comenzar a recibir eventos de entrada. Esta clase posee varios métodos que pueden ser sobreescritos:

**botonJugar = new TextButton(" jugar ", estiloBotones);**

**botonJugar.addListener(new InputListener() {**

**public boolean touchDown(InputEvent event, float x, float y,int pointer, int button) {**

**juego.reproducirSonido(sonidoBoton);**

**juego.setScreen(new SeleccionNivel(juego));**

**return true;**

**};**

**});**

En el ejemplo se le registra un InputListener a un TextButton (widget de Scene2d). En el listener se sobreescribe el método touchUp, evento propio del mouse o touchscreen. También se pueden sobreescribir otros métodos para estos dispositivos de entrada, como touchDragged, touchUp, enter, exit, mouseMoved, entre otros. También se pueden manejar eventos de teclado, como keyDown, keyUp, y keyTyped.

**SCENE2D.UI**

Para la creación de interfaces de usuario, el paquete scane2d.ui brinda widgets y otras clases creadas sobre scene2d.

Widget y Widgetgroup

Las clases Widget y WidgetGroup extienden de Actor y Group respectivamente. Los widgets de interfaz de usuario deberán extender a WidgetGroup si poseen actores hijos. Si no, de Widget.

Layout

Los widgets de UI no setean su medida y posición. Es el widget padre quien setea la medidas y posición de sus hijos. Los Widgets proveen medidas mínimas, preferidas, y máximas, que el padre usar como “pistas”.

Existen muchas clases para la creación de UI utilizando scene2d. Algunas son:

Table: Es un WidgetGroup que setea la medida y posición de sus hijos utilizando una tabla lógica, como las blas HTML.

Label: Muestra texto utilizando un BitmapFont y un color.

Image: Muestra un Drawable, que puede ser una textura, una región,un sprite, etc.

Button: Se trata de un botón vacío. Al extender de Table, se le pueden añadir otros widgets.

TextButton: Extiende de Button y contiene un label.

ScrollPane: Permite scrollear un widget hijo con scrollbars y/o arrastrando el mouse o el dedo.

Existen muchos otros widgets como por ejemplo ImageButton, CheckBox, ButtonGroup, TextField, Stack, List, SelectBox, Slider, SplitPane, Window, Touchpad, Tree, Dialog, entre otros.

Asset Manager

Es recomendado utilizar un AssetManager para administrar los assets de la aplicación. Aunque si es muy simple y si la carga de recursos no tarda mucho tiempo no es necesario.

Las ventajas que brinda el uso de la clase AssetManager son las siguientes:

La carga de la mayoría de los recursos se hace de manera asíncrona, por lo que se puede mostrar una pantalla que reaccione a la carga de los assets.

Si existen por ejemplo dos assets A y B que dependen de un asset C, C no será desechado hasta que A, B, y C sean desechados. También si se carga un asset muchas veces, ocupará memoria sólo una vez.

Se tiene un solo lugar donde se pueden almacenar todos los assets.

Para crear un AssetManager es muy simple:

**AssetManager adminAssets = new AssetManager();**

Para cargar los assets primero se debe llamar al método **load():**

**adminAssets.load("introduccion.png", Texture.class);**

**adminAssets.load("botones.atlas", TextureAtlas.class);**

**adminAssets.load("fuente\_negra.fnt", BitmapFont.class);**

**adminAssets.load("botonMenu.mp3", Sound.class);**

**adminAssets.load("musicaMenu.mp3",Music.class);**

Esas llamadas pondrán en una cola a los assets para ser cargados.

Para cargarlos, se debe llamar al método update() continuamente:

(getAdminAssets devuelve la instancia de AssetManager).

**public void render(float delta) {**

**if (juego.getAdminAssets().update())**

**((Game) Gdx.app.getApplicationListener()).setScreen(new MenuPrincipal(juego));**

**}**

Mientras que el método update() del AssetManager retorne false, seguirá cargando assets. Para consultar el porcentaje de carga se puede utilizar el método getProgress().

Para utilizar los assets cargados en el AssetManager:

**juego.getAdminAssets().get("ayuda1.png", Texture.class);**

Esto es, asumiendo que los assets fueron cargados. Si se desea consultar si un asset se cargó se puede hacer lo siguiente:

**If (juego.getAdminAssets().isLoaded("ayuda1.png"){**

**//hacer algo…**

**}**

También es simple desechar los assets utilizando el AssetManager:

**juego.getAdminAssets().unload("ayuda1.png");**

Es importante mencionar que los assets administrados por el AssetManager no deberían desecharse manualmente, sino con el método **unload()**.

Si se desean desechar todos de una vez, se puede llamar al método **clear()** o **dispose()** del AssetManager. La diferencia entre ellos es que el primero desecha los assets y remueve todos los assets en cola que no fueron cargados; mientras que el segundo también desecha el AssetManager.

**Consultas**

La interfaz Application provee varios métodos para generar consultas, como por ejemplo obtener el tipo de la aplicación, el consumo de energía, etc.

Obtener el tipo de aplicación

A veces es necesario saber en qué plataforma está ejecutándose la aplicación. El método

Application.getType() retorna la plataforma en la que se está ejecutando. En este fragmento de código del juego se puede ver:

**ApplicationType tipoApp= Application.getType();**

**if (tipoApp == ApplicationType.Android) {**

**mapa.update(deltaTime, Gdx.input.getAccelerometerX());**

**}**