Escuela
Politécnica
Superior

Interfaces de usuario en dispositivos móviles

iOS básico

Máster Universitario en Desarrollo de Software para Dispositivos Móviles

Otto Colomina Pardo

Departamento de Ciencia de la Computación e I.A.

Universitat d'Alacant
Universidad de Alicante

Table of Contents

Interfaz de usuario básico en iOS

En estos apuntes vamos a introducir los componentes básicos de la interfaz de usuario en aplicaciones iOS: las vistas, que son los componentes de la interfaz propiamente dicha y los controladores, que gestionan estas vistas.

Primero veremos cómo instanciar los controladores, que controlan las "pantallas" de nuestra aplicación, y cómo navegar de una pantalla a otra mediante el uso de *storyboards* o por código.

En cuanto a las vistas veremos las propiedades básicas y algunos controles de interfaz de usuario. Trataremos con más detalle un tipo especial de vista muy útil y habitual en aplicaciones móviles: las vistas de tabla, que nos permiten mostrar datos de manera organizada.

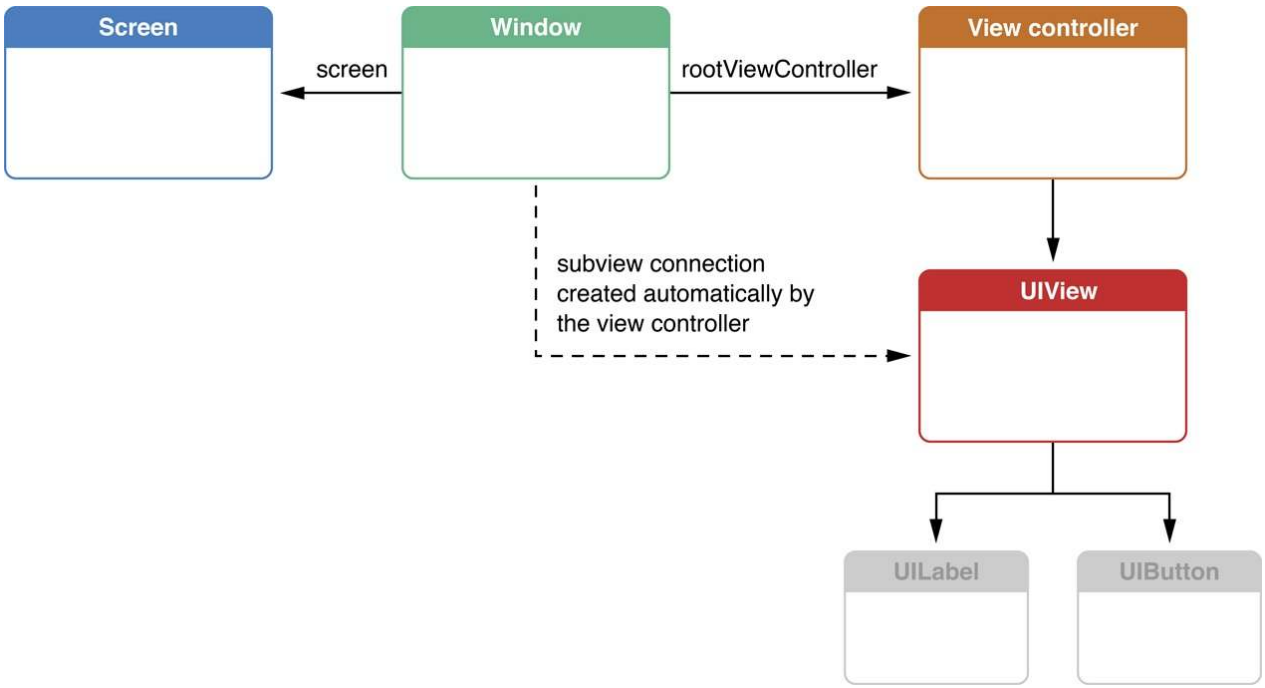
También veremos cómo adaptar la interfaz de usuario de modo que tenga un aspecto correcto independientemente de la orientación del dispositivo (horizontal o vertical) y de su resolución física. Para ello usaremos un mecanismo denominado *autolayout*.

Sesión 1: View Controllers

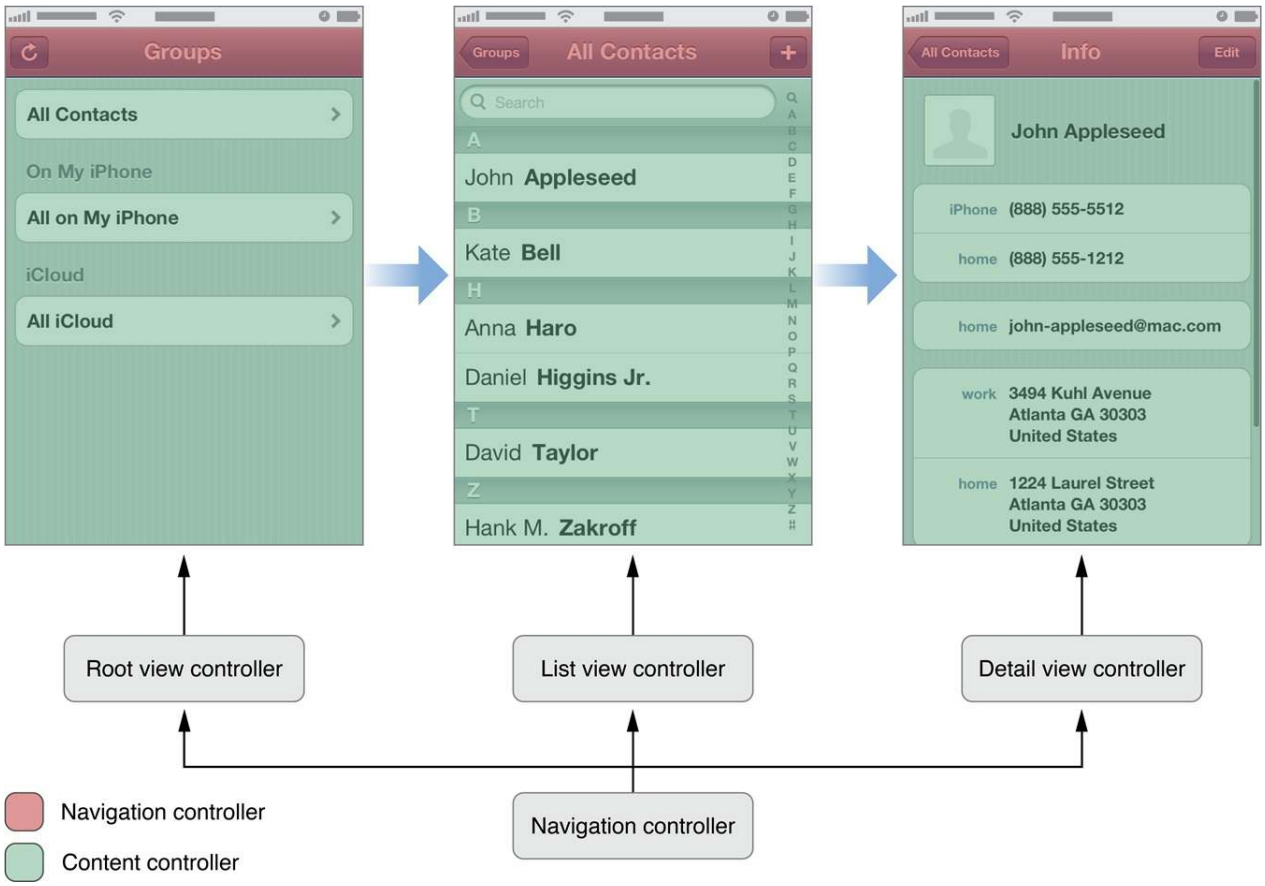
View controllers. Tipos básicos

Los *view controllers* son la C del MVC. Actúan como el “pegamento” que relaciona la vista con el modelo. El controlador es el lugar típico para poner el código que reacciona a los eventos del usuario. Por ejemplo qué hacer cuando se pulsa un botón.

En cada momento hay un *root view controller*, que es el “principal”. En los casos más simples, una pantalla de nuestra aplicación tendrá un único *view controller*. En general, un controlador puede tener a su vez controladores “hijos”, como veremos.



Hay dos tipos básicos de controladores: los que muestran directamente contenido (*content controllers*) y los que contienen otros contenedores (*container controllers*).



Cuando se usan *storyboards* no tenemos que preocuparnos de instanciar el controlador adecuado, el proceso es automático. También el controlador se asocia automáticamente a la vista que hemos definido en el *Interface Builder*.

Instanciar controladores y vistas

Podemos hacerlo de varias formas. De más sencilla a más compleja (pero también más flexible)

- **Gráficamente, con *storyboards***: tanto las vistas como el controlador están en el *storyboard*
  - **Gráficamente, con `.nib`** : en cada archivo `nib` guardamos una pantalla (con su jerarquía de vistas), pero no el controlador, que se crea por código
- **Totalmente por código**: tenemos que instanciar el controlador y en su método `loadView` crear la jerarquía de vistas que queremos que contenga (`UIView` , `UIButton` , lo que sea) y asignar la raíz de la jerarquía de vistas a `self.view` .

En los siguientes apartados vamos a ver cada una de estas posibilidades con algo más de detalle.

Storyboards

Desde Xcode 5 los *storyboards* son la forma recomendada por Apple de crear interfaces de usuario. Un *storyboard* contiene la representación gráfica de las “pantallas” (los controladores) que componen nuestra aplicación y de las relaciones entre ellas. Además el sistema se encarga automáticamente de moverse por las pantallas cuando sucedan determinados eventos, instanciando los controladores y las vistas automáticamente.

El *controller* inicial

En cada momento habrá un *view controller* inicial que es el que se muestra cuando se carga la aplicación. Se distingue visualmente porque tiene una flecha apuntando a él desde la izquierda:

Para **convertir un *view controller* en inicial**, teniéndolo seleccionado ir al icono de propiedades del área de `Utilities` y marcar sobre el *checkbox* `Is initial view controller`

También podemos arrastrar la flecha que indica que un controlador es el inicial desde el actual hasta el que queremos convertir en inicial.

Segues

Son las transiciones entre los *view controllers*. Podemos **crear un *segue* visualmente** con `Ctrl+Arrastrar` entre un elemento cualquiera de un view controller (por ejemplo un botón), que será el de *controller* de origen, y el *controller* destino. Se nos dará a **elegir el tipo de *segue*** en un menú contextual. Si es un *controller* de contenido el típico es `modal` (o también `popover` en iPad). El tipo `push` es para controladores de navegación, y el `custom` para escribir nuestro propio tipo.

Podemos **configurar las propiedades del *segue*** haciendo clic sobre él y yendo al icono de propiedades del área de `Utilities`. Aquí podemos cambiar el tipo y también la transición usada para navegar de una pantalla a otra.

Pasar datos de un *controller* a otro en un *segue*

Cuando se va a saltar de un *controller* a otro a través de un *segue*, se llama al método `prepareForSegue:sender:` del *controller* origen. Podemos sobreescribir este método para pasarle datos al *controller* destino. El primer parámetro va a instanciarse al *segue* y a partir de este podemos obtener una referencia al destino.

```
- (void) prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    //Suponemos que el controller destino es de la clase "ViewController2"
    ViewController2 *destino = [segue destinationViewController];
    //Suponemos que la clase "ViewController2"
    //tiene una @property NSString *texto
    destino.texto = @"Hola, soy el controller origen";
}
```

Volver atrás en un *segue*

Una cosa que no podemos hacer visualmente sin escribir algo de código es volver a la pantalla anterior cuando hemos seguido un *segue*. Para conseguirlo tenemos que hacer dos cosas:

La primera, implementar en el *controller* al que se vuelve un método que puede tener el nombre que deseemos pero debe devolver un `IBAction` y tener como único parámetro un `UIStoryboardSegue *`. Por ejemplo

```
- (IBAction)miUnwind:(UIStoryboardSegue *)segue {
    NSLog(@"Vuelta atrás!!");
}
```

Ahora en la pantalla del *storyboard* desde la que volvemos conectamos con `Ctrl+Arrastrar` entre el elemento de interfaz que debe producir el *unwind*, y el icono de `Exit` que aparece en la parte de arriba.

> Si intentamos hacer esta operación de `Ctrl+Arrastrar` sin haber implementado el método anterior, veremos que no tiene efecto

En el método del *unwinding*, nótese que podemos usar el parámetro, que es el *segue*, para obtener el `destinationController`, que ahora será el *controller* al que volvemos. También podemos acceder al *controller* desde el que volvemos en la propiedad `sourceController`.

Finalmente, decir que cuando se produce un *unwind*, el controlador desde el que se vuelve también recibe una llamada a `prepareForSegue:sender`, método que podemos sobreescribir si queremos aprovechar para realizar alguna operación antes de volver.

NIBs

Un archivo NIB (o `.xib`, en un momento veremos la diferencia) contiene la jerarquía de vistas asociada a un determinado *view controller*, pero normalmente no se crea de manera manual, sino visualmente con el Interface Builder. De hecho, el nombre significa “NeXT Interface Builder”, referenciando la famosa plataforma NeXTSTEP de la que hereda y es deudora Cocoa.

Hasta que apareció iOS 5 los NIB eran la forma habitual de crear interfaces de usuario, pero por defecto las últimas versiones de Xcode (desde la 5, correspondiéndose con iOS7) usan *storyboards*. Nótese que un NIB contiene únicamente “una pantalla” de nuestra aplicación y que por tanto es responsabilidad del desarrollador cambiar de un controlador a otro y cargar el NIB correspondiente conforme se va navegando.

Un archivo .xib, que es lo que vemos en la lista de archivos de proyecto en Xcode, es básicamente un NIB serializado en forma de XML, lo que podemos comprobar haciendo *clic* sobre él con el botón derecho en Xcode y seleccionando `Open as` > `Source code`.

En Xcode podemos crear un NIB de dos formas:

- Crear un controller y automáticamente un NIB asociado
- Crear directamente el NIB y luego asociarle manualmente un controller

En la primera de las posibilidades nos iríamos a crear la nueva clase del *controller* (una `cocoa touch class`). Especificaríamos que hereda de `UIViewController` y marcaríamos la casilla de `Also create XIB File`.

Como el controlador y el NIB están asociados por defecto, para cargar la pantalla con la jerarquía de vistas basta con cargar el controlador:

```
MiViewController *mvc = [[MiViewController alloc] init];
//pasamos a este controller
self.window.rootViewController = mvc;
```

En el segundo caso (asociación “manual”), cuando inicializamos el controlador debemos especificar qué NIB debe cargar

```
EjemploViewController *evc =
  [[EjemploViewController alloc] initWithNibName:@"ejemplo" bundle:nil];
self.window.rootViewController = evc;
```

Ejercicios de *view controllers*

Vamos a hacer una aplicación que vamos a llamar “Pioneras”, y que nos dará datos de algunas mujeres pioneras de la informática. La aplicación tendrá una pantalla principal en la que aparecerán sus imágenes, y haciendo *tap* sobre cada una podremos ir a las pantallas secundarias donde se nos dará más información.

Realizar la estructura básica de la aplicación

1. En la carpeta `sesion1/recursos/` de las plantillas tenemos las imágenes de las tres pioneras: Ada Lovelace, Grace Hopper y Barbara Liskov, que como siempre **arrastraremos al** `images.xcassets`. También tenemos los textos sobre ellas que se mostrarán en las pantallas secundarias.
2. Crea tres botones en la pantalla principal, y para cada uno de ellos en lugar de texto vamos a usar como imagen de fondo la de cada mujer. Al final cada botón debería ocupar todo el ancho de la pantalla y un tercio del alto.
3. Arrastra un nuevo “view controller” al storyboard (una “pantalla” nueva), que será el que aparezca cuando se pulse en el primero de los botones (el de Ada Lovelace). Inserta un campo de texto de varias líneas (*text view*) y copia en él el contenido de `sesion1/recursos/lovelace.txt`
4. Ahora **establece el *segue* entre las dos pantallas**: haz `Ctrl+Arrastrar` desde el primero de los botones con la imagen de Ada Lovelace hasta la segunda pantalla. En el menú contextual elige el tipo `modal`, **ya que los otros no funcionarán**.
    - Si haces *clic* en el *segue* y vas al *Attribute inspector* puedes cambiar las propiedades, pero tal como está hecha la aplicación solo va a tener efecto la `transition`. Pon el valor que quieras, salvo `partial curl` que puede dar problemas a la hora de volver atrás en el *segue*.
    - Ejecuta el proyecto para comprobar que funciona lo que has hecho, aunque *todavía no puede volver atrás desde la pantalla secundaria*
5. Implementa la opción de **volver atrás** de la secundaria a la principal
    - Crea un botón “atrás” en la secundaria y colócalo en la parte de arriba (para que no lo tape el teclado *on-screen* si aparece)
    - En el *controller* destino crea un método para que funcione el *unwinding* (no hace falta que haga nada, solo que exista)

```
- (IBAction)retornoDeSecundaria:(UIStoryboardSegue*)segue {
}
```

- Con `Ctrl+Arrastrar` conecta el botón “atrás” con el icono de “Exit” de la parte superior del *controller*
    - Ejecuta el proyecto y comprueba que puedes volver atrás desde la pantalla secundaria
- Repite lo que has hecho en el caso de Ada Lovelace para las otras dos mujeres, creando las pantallas secundarias y la navegación adelante y atrás.

Comunicar un *controller* con otro

Es un poco redundante tener tantas pantallas secundarias cuando en realidad lo único que cambia es el texto a mostrar. Valdría con una sola secundaria en la que cambiáramos dinámicamente dicho texto. Vamos a implementarlo así.

Guardar la versión actual del proyecto para que no se pierda

Lo primero es guardar una copia del estado actual del proyecto. Hay dos posibilidades:

- con `git` podemos crear una etiqueta o *tag* que nos marque la versión actual del proyecto para poder recuperarla luego. No obstante Xcode no nos permite gestionar *tags*, por lo que, **tras asegurarnos de que hemos hecho commit y push de todos los cambios actuales**, haríamos desde la terminal (estando dentro del repositorio):

```
git tag v1.0  (etiquetamos el commit actual)
git push origin v1.0 (subimos la etiqueta al repositorio remoto)
```

- Otra posibilidad es crear una copia manual de los ficheros del proyecto y guardarlo en una carpeta `v1.0` de las plantillas

Ahora podéis eliminar los segues y las pantallas secundarias, es mejor crearlos de nuevo.

Crear la nueva interfaz

- Crea de nuevo una pantalla secundaria con un campo de texto de varias líneas
- Con `Ctrl+arrastrar` podemos crear un *segue* desde cada uno de los botones hasta la pantalla. Habrán tres *segues* que lleguen a la misma, no debería ser problema.
- Añádele a la pantalla el botón de “atrás” y conéctalo con el icono de “exit”. El código necesario para el *unwinding* (método `retornoDeSecundaria`) ya debería estar en el `ViewController.m`
- Comprueba que la navegación funciona correctamente yendo adelante y atrás

Crear un controlador personalizado para la pantalla secundaria

Si en la parte derecha de la pantalla miras el *identity inspector* verás que el controlador de la pantalla secundaria es un tipo propio de Cocoa, el `UIViewController`. Vamos a cambiarlo por uno propio

1. Crea una nueva clase de Cocoa Touch, (File> New > File…, plantilla “cocoa touch class”). En la segunda pantalla del asistente dale a la clase el nombre `SecundarioViewController` y haz que sea una subclase de `UIViewController`. Deja sin marcar la opción de crear el .XIB
2. En el *storyboard*, selecciona el *controller* de la pantalla secundaria (es mejor que lo hagas pulsando en el primero de los iconos que aparecen en la parte superior) 
3. Una vez seleccionado, ve al *identity inspector* en el área de `Utilities` y en el apartado de `Custom class` selecciona como clase la que has creado, `SecundarioViewController`

Añadirle un *outlet* al controlador secundario

Tienes que añadir un *outlet* al campo de texto para que su contenido se pueda cambiar desde el controlador secundario. Hazlo como habitualmente, con ctrl+arrastrar entre el campo y el `SecundarioViewController.h`, en el `assistant editor`.

Hacer que el texto cambie según el botón pulsado

- Lo primero es añadir físicamente los ficheros `*.txt` con los textos al proyecto para que se puedan cargar dinámicamente por código. Pulsa con el botón derecho sobre el proyecto y selecciona `Add files to Pioneras`. Selecciona los tres `.txt`, que se añadirán al proyecto
- Para que le podamos decir al controlador secundario qué fichero tiene que abrir, vamos a crear una `@property` en el `SecundarioViewController` llamada `nomFich` de tipo `NSString*`

  //En el SecundarioViewController.h @property NSString *nomFich;

- Para establecer una asociación sencilla entre cada segue y los datos a mostrar puedes usar el identificador del *segue*. Haz clic sobre él y en el `Attributes inspector` cambia su `identifier`, respectivamente por `lovelace`, `hopper` y `liskov`

- ahora en la clase `ViewController`, que es el controlador de la pantalla principal, puedes implementar el `prepareForSegue:sender`

```
//En el ViewController.m
-(void)prepareForSegue:(UIStoryboardSegue *)segue sender:(id)sender {
    SecundarioViewController *svc = segue.destinationViewController;
    //El fichero a cargar se corresponde con el identificador del segue
    svc.nomFich = segue.identifier;
}
```

- Finalmente, en el `viewDidLoad` del `SecundarioViewController` puedes acceder a la propiedad `self.nomFich`, cargar el texto del fichero y mostrarlo en el campo de texto.

```
//Sacamos el path completo el fichero
NSString *filePath = [[NSBundle mainBundle] pathForResource:self.nomFich                                    ofType:@"txt"];
NSError *error;
NSString *textoFich;
if (filePath) {
    //Leemos el fichero y guardamos su contenido en un NSString*
    textoFich = [NSString stringWithContentsOfFile:filePath
                        encoding:NSUTF8StringEncoding
                        error:&error];
//HAY QUE CAMBIAR self.campoTexto POR COMO HAYAIS LLAMADO AL OUTLET!!!!!
if (!error) {
    //Si todo es OK, mostramos la cadena en el campo de texto
    self.campoTexto.text = textoFich;
}
```

Sesión 2: Vistas

Creación de vistas por código

Hasta ahora hemos visto como crear la interfaz visualmente con Xcode, pero todo lo que se puede hacer con dicha herramienta se puede hacer también de forma programática, ya que lo único que hace el entorno es crear objetos de la API de Cocoa Touch (o definidos por nosotros) y establecer algunas de sus propiedades.

Ventanas

Las aplicaciones iOS tienen una única ventana. Podríamos crear la ventana principal desde nuestro código instanciando un objeto de tipo `UIWindow` e indicando el tamaño del marco de la ventana:

```
UIWindow *window = [[UIWindow alloc]
    initWithFrame: [[UIScreen mainScreen] bounds]];
```

Lo habitual será inicializar la ventana principal con un marco del tamaño de la pantalla, tal como hemos hecho en el ejemplo anterior.

Para mostrar la ventana en pantalla debemos llamar a su método makeKeyAndVisible (esto se hará normalmente en el método `application:didFinishLaunchingWithOptions:` del delegado de UIApplication). Esto se hará tanto cuando la ventana ha sido cargada de un NIB como cuando la ventana se ha creado de forma programática.

```
[self.window makeKeyAndVisible];
```

Es posible que necesitemos acceder a esta ventana, por ejemplo para cambiar la vista que se muestra en ella. Hacer esto desde el delegado de UIApplication es sencillo, porque normalmente contamos con la propiedad `window` que hace referencia a la pantalla principal, pero acceder a ella desde otros lugares del código podría complicarse. Para facilitar el acceso a dicha ventana principal, podemos obtenerla a través del singleton `UIApplication`:

```
UIWindow *window = [[UIApplication sharedApplication] keyWindow];
```

Si podemos acceder a una vista que se esté mostrando actualmente dentro de la ventana principal, también podemos obtener dicha ventana directamente a través de la vista:

```
UIWindow *window = [vista window];
```

Vistas genéricas

También podemos construir una vista creando un objeto de tipo `UIView`. En el inicializador deberemos proporcionar el marco que ocupará dicha vista. El marco se define mediante el tipo `CGRect` (se trata de una estructura, no de un objeto). Podemos crear un nuevo rectángulo con la macro `CGRectMake`. Por ejemplo, podríamos inicializar una vista de la siguiente forma:

```
UIView *vista = [[UIView alloc]
    initWithFrame:CGRectMake(0,0,100,100)];
```

Como alternativa, podríamos obtener el marco a partir del tamaño de la pantalla o de la aplicación:

```
// Marco sin contar la barra de estado
CGRect marcoVista = [[UIScreen mainScreen] applicationFrame];
// Limites de la pantalla, con barra de estado incluida
CGRect marcoVentana = [[UIScreen mainScreen] bounds]
```

Normalmente utilizaremos el primero para las vistas que queramos que ocupen todo el espacio disponible en la pantalla, y el segundo para definir la ventana principal. Hemos de destacar que `UIWindow` es una subclase de `UIView`, por lo que todas las operaciones disponibles para las vistas están disponibles también para las ventanas. Las ventanas no son más que un tipo especial de vista.

Las vistas (`UIView`) también nos proporcionan una serie de métodos para consultar y modificar la jerarquía. El método básico que necesitaremos es `addSubview` , que nos permitirá añadir una subvista a una vista determinada (o a la ventana principal):

```
//En el application delegate
[self.window addSubview: vista];
```

Con esto haremos que la vista aparezca en la pantalla.

También podemos añadir una vista usando la propiedad `view` del *view controlller* "activo". Esta propiedad representa la vista "raíz" de la jerarquía de vistas manejadas por él.

```
//En el view controller
[self.view addSubview: vista];
```

Otra posibilidad es sobreescribir el `loadView` del controlador, que es el método encargado por defecto de crear las vistas, usualmente desde un NIB o *storyboard*, y en él crear las vistas necesarias, asignando la vista raíz a `self.view` .

Podemos eliminar una vista enviándole el mensaje `removeFromSuperview` (se le envía a la vista hija que queremos eliminar). Podemos también consultar la jerarquía con los siguientes métodos:

- `superview` : Nos da la vista padre de la vista destinataria del mensaje.
- `subviews` : Nos da la lista de subvistas de una vista dada.
- `isDescendantOfView:` Comprueba si una vista es descendiente de otra.

Como vemos, una vista tiene una lista de vistas hijas. Cada vista hija tiene un índice, que determinará el orden en el que se dibujan. El índice 0 es el más cercano al observador, y por lo tanto tapará a los índices superiores. Podemos insertar una vista en un índice determinado de la lista de subvistas con `insertSubview:atIndex:` .

Puede que tengamos una jerarquía compleja y necesitemos acceder desde el código a una determinada vista por ejemplo para inicializar su valor. Una opción es hacer un *outlet* para cada vista que queramos modificar, pero esto podría sobrecargar nuestro objeto de *outlets*. También puede ser complejo y poco fiable el buscar la vista en la jerarquía. En estos casos, lo más sencillo es darle a las vistas que buscamos una etiqueta (*tag*) mediante la propiedad `Tag` del inspector de atributos (debe ser un valor entero), o asignando la propiedad `tag` de forma programática. Podremos localizar en nuestro código una vista a partir de su etiqueta mediante `viewWithTag` . Llamando a este método sobre una vista, buscará entre todas las subvistas aquella con la etiqueta indicada:

```
UIView *texto = [self.window viewWithTag: 1];
```

> La jerarquía de vistas de una pantalla determinada de nuestra aplicación puede llegar a ser muy compleja. Es por eso que en Xcode 6 se ha añadido una opción que nos permite mostrar un "despiece" visual en 3D de las vistas que componen la pantalla actual. Dicha opción está disponible en `Debug > View Debugging` .En modo texto podemos usar la propiedad `recursiveDescription` para imprimir la descripción textual de las vistas que contiene una vista dada.

Propiedades de una vista

A continuación vamos a repasar las propiedades básicas de las vistas, que podremos modificar tanto desde Xcode como de forma programática.

Disposición

Entre las propiedades más importantes en las vistas encontramos aquellas referentes a su disposición en pantalla. Hemos visto que tanto cuando creamos la vista con Xcode como cuando la inicializamos de forma programática hay que especificar el **marco** que ocupará la vista en la pantalla.

Cuando se crea de forma visual, el marco se puede definir pulsando con el ratón sobre los márgenes de la vista y arrastrando para así mover sus límites. En el código estos límites se especifican mediante el tipo `CGRect`, en el que se especifica posición `(x,y)` de inicio, y el ancho y el alto que ocupa la vista. Estos datos se especifican en el sistema de coordenadas de la supervista.

El sistema de coordenadas tiene su origen en la esquina superior izquierda. Las coordenadas no se dan en *pixels*, sino en **puntos**, una medida que nos permite independizarnos de la resolución en pixels de la pantalla. Las coordenadas en puntos son reales, no enteras.

En los modelos de iPhone/iPod Touch de 3.5'' la resolución de pantalla en puntos es de 320x480 (aun en los de *retina display*, que tiene un número de pixels mucho mayor). Los dispositivos de 4'' usan una resolución de 320x568 puntos. El iPhone 6 devuelve 375x667 y el 6 plus 736x414. Podéis consultar una tabla muy completa con muchos más datos

> Otros frameworks de iOS definen sistemas de coordenadas distintos. Los de gráficos (Core Graphics y OpenGL ES) ponen el origen en la esquina inferior izquierda con el eje Y apuntando hacia arriba.

Algunos ejemplos de cómo obtener la posición y dimensiones de una vista:

```
// Limites en coordenadas locales
// Su origen siempre es (0,0)
CGRect area = [vista bounds];
// Posición del centro de la vista en coordenadas de su supervista
CGPoint centro = [vista center];
// Marco en coordenadas de la supervista
CGRect marco = [vista frame]
```

A partir de `bounds` y `center` podremos obtener `frame`

Sin embargo, en muchas ocasiones nos interesa que el tamaño no sea fijo sino que se adapte al área disponible. De esta forma nuestra interfaz podría adaptarse de forma sencilla a distintas orientaciones del dispositivo (horizontal o vertical) o a distintas resoluciones de la pantalla. Esto lo podemos conseguir mediante el uso de la tecnología de *autolayout*, que calcula de manera automática el *frame* de cada vista basándose en un conjunto de restricciones. Veremos esta tecnología en sesiones posteriores.

Transformaciones

Podemos también aplicar una transformación a las vistas, mediante su propiedad `transform`. Por defecto las vistas tendrán aplicada la transformación identidad `CGAffineTransformIdentity`.

La transformación se define mediante una matriz de transformación 2D de dimensión 3x3. Podemos crear transformaciones de forma sencilla con macros como `CGAffineTransformMakeRotation` o `CGAffineTransformMakeScale`.

> Si nuestra vista tiene aplicada una transformación diferente a la identidad, su propiedad `frame` no será significativa. En este caso sólo deberemos utilizar `center` y `bounds`.

Otras propiedades

En las vistas encontramos otras propiedades que nos permiten determinar su color o su opacidad. En primer lugar tenemos `backgroundColor`, con la que podemos fijar el color de fondo de una vista. En el inspector de atributos (sección `View`) podemos verlo como propiedad `Background`. El color de fondo puede ser transparente, o puede utilizarse como fondo un determinado patrón basado en una imagen.

De forma programática, el color se especifica mediante un objeto de clase `UIColor`. En esta clase podemos crear un color personalizado a partir de sus componentes (rojo, verde, azul, alpha), o a partir de un patrón.

Por otro lado, también podemos hacer que una vista tenga un cierto grado de transparencia, o esté oculta. A diferencia de `backgroundColor`, que sólo afectaba al fondo de la vista, con la propiedad `alpha`, de tipo `CGFloat`, podemos controlar el nivel de transparencia de la vista completa con todo su contenido y sus subvistas. Si una vista no tiene transparencia, podemos poner su propiedad `opaque` a `YES` para así optimizar la forma de dibujarla. Esta propiedad sólo debe establecerse a `YES` si la vista llena todo su contendo y no deja ver nada del fondo. De no ser así, el resultado es impredecible. Debemos llevar cuidado con esto, ya que por defecto dicha propiedad es `YES`.

Por último, también podemos ocultar una vista con la propiedad `hidden`. Cuando hagamos que una vista se oculte, aunque seguirá ocupando su correspondiente espacio en pantalla, no será visible ni recibirá eventos.

Controles básicos de interfaz de usuario

A lo largo de los ejemplos que hemos ido haciendo en las sesiones anteriores ya hemos probado bastantes de los controles básicos de interfaz de usuario que nos proporciona iOS: botones, etiquetas, imágenes, campos de texto,… Vamos a ver aquí algunas de las características de los controles, aunque solo vamos a dar unas pinceladas, ya que una descripción exhaustiva de cada propiedad sería imposible y tediosa. Se remite al lector a la documentación de Apple, en concreto el UIKit User Interface Catalog, excelente y bastante exhaustivo.

> Aunque aquí hablemos de controles indistintamente para referirnos a las etiquetas, botones, … en realidad este término tiene un significado más preciso en iOS. La clase `UIControl` es de la que heredan los controles más "interactivos" como los botones, mientras que las etiquetas lo hacen de `UIView` (no obstante todos los `UIControl` son también vistas ya que a su vez esta clase hereda de `UIView`).

Campos de texto

Un campo de texto nos proporciona un espacio donde el usuario puede introducir y editar texto. Se define en la clase `UITextField`, y pertenece a un grupo de vistas denominados controles, junto a otros componentes como por ejemplo los botones. Esto es así porque permiten al usuario interactuar con la aplicación. No heredan directamente de `UIView`, sino de su subclase `UIControl`, que incorpora los métodos para tratar eventos de la interfaz mediante el patrón *target-action* como hemos visto anteriormente.

Sus propiedades se pueden encontrar en la sección `Text Field` del inspector de atributos. Podremos especificar un texto por defecto (`Text`), o bien un texto a mostrar sombreado en caso de que el usuario no haya introducido nada (`Placeholder Text`). Esto será útil por ejemplo para dar una pista al usuario sobre lo que debe introducir en dicho campo.

Si nos fijamos en el inspector de conexiones del campos de texto, veremos la lista de eventos que podemos conectar a nuestra acciones. Esta lista de eventos es común para cualquier control. En el caso de un campo de texto por ejemplo nos puede interesar el evento `Value Changed`.

Botones

Al igual que los campos de texto, los botones son otro tipo de control (heredan de `UIControl`). Se definen en la clase `UIButton`, que puede ser inicializada de la misma forma que el resto de vistas.

Si nos fijamos en el inspector de atributos de un botón (en la sección `Button`), vemos que podemos elegir el tipo de botón (atributo `Type`). Podemos seleccionar una serie de estilos prefedinidos para los botones, o bien darle un estilo propio (`Custom`).

El texto que aparece en el botón se especifica en la propiedad `Title`, y podemos configurar también su color,

sombreado, o añadir una imagen como icono.

En el inspector de conexiones, el evento que utilizaremos más comúnmente en los botones es `Touch Up Inside`, que se producirá cuando levantemos el dedo tras pulsar dentro del botón. Este será el momento en el que se realizará la acción asociada al botón.

Alertas

Se usan para informar al usuario de eventos importantes. Pueden simplemente informar de algo o además pedir al usuario que elija uno entre varios cursos de acción. No se crean gráficamente en Xcode sino por código

```
UIAlertView *alert = [[UIAlertView alloc]
                      initWithTitle:@"Saludo"
                      message:@"Hola usuario"
                      delegate:self
                      cancelButtonTitle:@"OK"
                      otherButtonTitles: nil];
[alert show]
```

Como mínimo tendremos un botón (`cancelButtonTitle`), para que la alerta desaparezca.

> Aunque se llame `cancelButtonTitle` no quiere decir que sea el botón de cancelar, de hecho habitualmente será "Aceptar" u "OK".

Podemos tener más botones, como una lista terminada en `nil`. No obstante, las "iOS Human Interface Guidelines" recomiendan como máximo dos, y en caso que necesitemos más usar un *"Action Sheet"*.

El `delegate` es el objeto que se ocupará de qué hacer cuando la alerta se hace desaparecer o qué curso tomar en función del botón elegido. Debe implementar el protocolo `UIAlertViewDelegate`. Por ejemplo `alertView:didDismissWithButtonIndex:` se envía cuando desaparece la alerta y nos dice qué número de botón se ha pulsado (0 el `cancelButtonTitle` y el resto por el orden de definición comenzando en 1).

```
-(void) alertView:(UIAlertView *)alertView didDismissWithButtonIndex:(NSInteger)buttonIndex {
   NSLog(@"Se ha pulsado el boton nº %d", buttonIndex);
}
```

Action Sheets

Dan a los usuarios la posibilidad de ofrecer datos adicionales ante una acción a realizar. Por ejemplo confirmar o cancelar la información.

> Las aparición de una alerta suelen ser inesperada para el usuario. Sin embargo el usuario sabe cuándo es probable que aparezca un Action Sheet. Por ejemplo cuando realiza una acción destructiva como eliminar una foto.

No se crean gráficamente sino por código. El API es prácticamente idéntico al de las alertas.

> Dese iOS 8, `UIAlertController` uniformiza el interfaz de `UIAlertView` y `UIActionSheet`. Ver por ejemplo http://nshipster.com/uialertcontroller/

Teclado en pantalla

Cuando un campo de texto adquiere el foco porque el usuario hace *tap* sobre él, automáticamente aparece el teclado software *on screen*. El problema es que por defecto no desaparece salvo que escribamos algo de código.

Si queremos que desaparezca cuando pulsamos sobre la tecla de "Aceptar" tenemos que escribir un *action* que responda al evento `Did end on exit` del campo de texto. En teoría dentro de este *action* debemos hacer que el campo

deje de ser el *first responder* para que el teclado deje de mostrarse

```
- (IBAction)pulsadoIntro:(id)sender {
   [sender resignFirstResponder];
}
``

> Aunque en muchas fuentes aparece el método del `resignFirstResponder` como solución, curiosamente *parece bastar con tener

Por desgracia, no todos los tipos de teclado en pantalla tienen un botón de “intro” y por tanto no disparan el evento `Did end on ex

```objectivec
-(void) touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {
NSLog(@"touch en la pantalla!!");
  //Suponiendo que al campo de texto le hayamos dado el tag=1
  [[self.view viewWithTag:1] resignFirstResponder];
}
```

Ejercicios de vistas

Creación de ventana/controlador/vista de modo manual

Ya hemos visto que por defecto las aplicaciones de Xcode usan el *storyboard*. Este *storyboard* lo carga automáticamente el Application Delegate, haciendo además las siguientes tareas:

- Crear una ventana con el tamaño de la pantalla
- Cargar el controlador asociado a la pantalla inicial, y asignarle a su propiedad `view` las subvistas de esta pantalla
- Hacer visible la ventana, que ahora contiene al controlador

Vamos a hacer lo mismo pero de modo manual, para comprender un poco mejor todo el proceso

- Cread un nuevo proyecto en la carpeta `sesion2` de las plantillas llamado `VistaManual`
- Lo primero es desactivar el *storyboard*. Para ello en el navegador de Xcode hacemos clic sobre el icono del proyecto. Aparecerán los distintos apartados de configuración. Elegimos `info` y eliminamos una propiedad que pone “Main Storyboard name” pulsando sobre el botón de ‘-‘. Ahora el proyecto ya no usa el storyboard
- En el `applicationDidFinishLaunching:withOptions` del AppDelegate tenemos que hacer lo siguiente

```
self.window = [[UIWindow alloc]
               initWithFrame: [[UIScreen mainScreen] bounds]];
self.window.rootViewController = [[ViewController alloc] init];
UIButton *boton = [[UIButton alloc] init];
[self.window makeKeyAndVisible];
```

- No obstante, en pantalla no se verá nada, ya que la vista del `ViewController` está vacía. Probad a hacer que aparezca algo para comprobar que todo funciona:
  - Tened en cuenta que la vista es `self.window.rootViewController.view`
  - cambiad el color, propiedad `backgroundColor` de la vista. Para especificar color lo más sencillo es usar una serie de métodos de clase (estático) de `UIColor` : `[UIcolor greenColor]` , `[UIcolor redColor]` ,…
  - cread un botón (instanciad un objeto de la clase UIButton*), especificando el título (método `setLabel:forState`) y unas dimensiones

```
[boton setTitle:@"Soy un botón manual" forState:UIControlStateNormal];
[boton setFrame:CGRectMake(100,100,100,40)];
```

Uso de controles de interfaz de usuario

En un nuevo proyecto, llamado `Controles` , probad el uso de algunos de los controles de interfaz de usuario de iOS que
```

todavía no hemos visto. Para cualquier duda, consultad la documentación de Xcode o la referencia de controles de UIKit

- Cread un *slider* cuya escala vaya desde 0 hasta 100, y un *label* al lado. Al mover el slider, el *label* debería cambiar para reflejar su valor actual.

    - Cread un *outlet* para poder manipular la etiqueta
    - Crear un "action" con `ctrl+arrastrar` desde el *slider* al código de `ViewController.m` . En este *action* podéis obtener el valor actual del *slider* (propiedad `value` del parámetro `sender` y fijarlo como texto de la etiqueta. Tened en cuenta que el valor del slider es un número y el texto es un `NSString*` , podéis usar `[NSString stringWithFormat]` para convertir como ya hemos hecho varias veces.
- cread un botón que cuando se pulse aparezca un `UIActionSheet` dando varias opciones con texto inventado por vosotros. Detectad qué opción se ha mostrado e imprimid un mensaje con `NSLog` indicándolo. Tendréis que consultar la documentación de `UIActionSheet` . Mirad el apartado "Behavior of action sheets". Básicamente la idea es que al crear el *action sheet* se especifica qué objeto es su "delegate" (lo más sencillo es que sea el controller) y este tiene que

    - Especificar que implementa el protocolo `UIActionSheetDelegate`
    - Implementar el método `actionSheet:clickedButtonAtIndex:` que nos dirá el número de opción elegida, comenzando en 0.

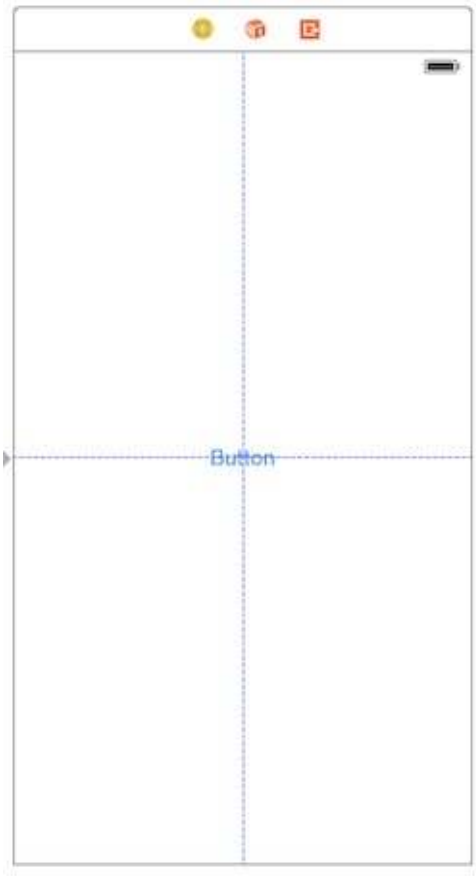
Sesión 3: Autolayout

Introducción

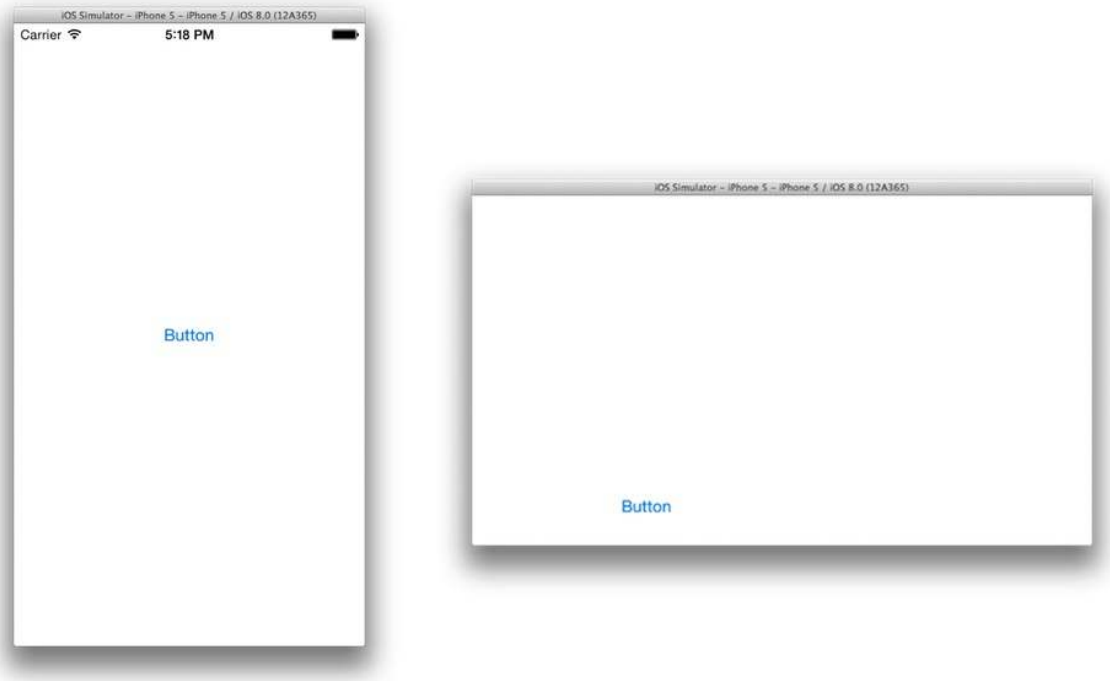
El problema de colocar los elementos de la interfaz en unas coordenadas fijas es que si rota la pantalla o cambiamos de dispositivo, el interfaz no se va adaptar adecuadamente, ya que las dimensiones han cambiado y las coordenadas antes especificadas ahora pueden no tener sentido.

> Para que la previsualización del interfaz en Xcode tenga el mismo aspecto que las figuras, hay que desmarcar la casilla “Use size classes” del “File Inspector” (Área a la derecha de la pantalla, primer icono)

Por ejemplo, supongamos que queremos centrar un botón en la pantalla, tanto vertical como horizontalmente. En principio parece que basta con moverlo con el ratón hacia el centro. En el momento que el botón está centrado, aparecen unas guías punteadas que nos lo indican.



Si ejecutamos la aplicación en el simulador veremos que efectivamente está centrado. Pero si rotamos la pantalla (tecla Cmd-Flecha izquierda) podremos ver que cuando cambia la resolución no es así.



Necesitamos algún sistema que adapte automáticamente las dimensiones de los componentes de la interfaz a la resolución actual. En iOS ese sistema es **Autolayout**. Es un sistema declarativo y basado en restricciones. El sistema usa las restricciones especificadas para calcular automáticamente el *frame* de cada vista de la interfaz, y adaptarlo a las dimensiones actuales de la ventana.

Manejo de restricciones con el Interface Builder

Para especificar qué aspecto queremos que tenga la interfaz independientemente de la resolución hay que añadir **restricciones**. Básicamente las hay de dos tipos:

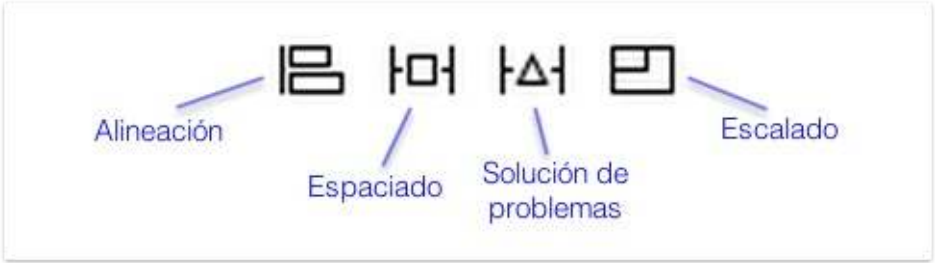
- **De alineación** (*align*): por ejemplo queremos que un botón esté centrado horizontalmente o verticalmente en su contenedor. O que varios componentes estén alineados entre sí.
- **De espaciado** (*pin*): por ejemplo queremos que entre un componente y otro, o entre un componente y el borde izquierdo de la pantalla haya un espacio vacío. Aquí también se incluirían las restricciones de tamaño de un componente individual. (fijar el ancho, el alto,…)

Hay dos formas de añadir restricciones en XCode: "haciendo clic y arrastrando" con el ratón o bien a través de la barra de herramientas de AutoLayout.

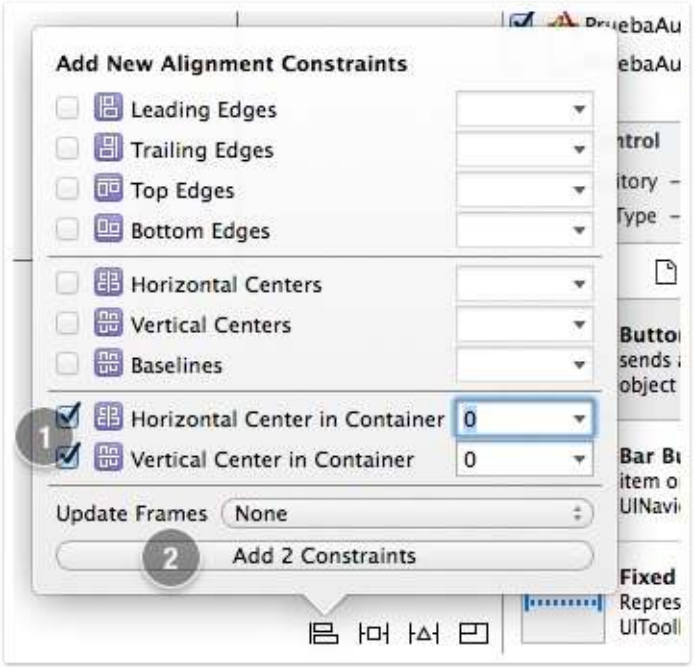
> Cuando se añaden restricciones de espaciado con respecto al borde superior e inferior de la pantalla, puede verse que en realidad no se están referenciando los bordes sino lo que Xcode llama `top layout guide` y `bottom layout guide`. La verdadera utilidad de estas guías es que se "mueven automáticamente" para dejar espacio a las barras de navegación y de botones que veremos cuando usemos *navigation controllers* y *tab bar controllers*, asegurándonos así de que dichas barras no tapan a nuestras vistas.

Añadiendo restricciones con botones/menús

En la parte inferior derecha del *storyboard* hay una barra de botones específicos para autolayout



Si queremos conseguir que funcione el ejemplo anterior en el que queríamos centrar horizontal y verticalmente el botón, pulsamos sobre el icono de `Align` (el primero), marcamos las casillas de `Horizontal center in container` y `Vertical center in container` y pulsamos sobre el botón que ahora pondrá `Add 2 constraints` para hacer efectivas las restricciones.



Las líneas de guía, que antes aparecían punteadas, ahora serán continuas indicando que ahora son restricciones de autolayout. Dichas restricciones podemos verlas en varios sitios de Xcode:

- En el área de `Document outline`, que es accesible pulsando sobre el icono %2018.03.41.png) que aparece en la parte inferior izquierda del *storyboard*. Aquí podemos ver un "árbol" desplegable con las restricciones.
  - Si hacemos clic sobre una restricción, en el área de `Utilities` de la derecha de la pantalla, dentro del `Size inspector` (el pequeño icono con una regla %2018.09.58.png)) aparecerán sus propiedades, que podemos editar (luego veremos qué significan).
  - Si hacemos clic sobre una restricción y pulsamos la tecla `Backspace` se eliminará.

- Directamente en el `Size inspector` aparece una lista de restricciones. Cada una tiene un botón `Edit` para cambiar sus propiedades.

Las mismas operaciones también las tenemos disponibles en la opción `Editor` del menú principal, a través de los submenús `Align` y `Pin`.

Añadiendo restricciones con el ratón

Esta forma es algo más ágil que la anterior pero requiere de cierta práctica. Cuando queremos establecer una restricción entre dos elementos **arrastramos de uno a otro manteniendo pulsada la tecla `Ctrl`** (igual que para crear un *outlet* o un *action*). Cuando soltamos el botón del ratón, aparece un menú contextual donde elegir la restricción. Las opciones disponibles en el menú dependen de la dirección y sentido en que se haya arrastrado:

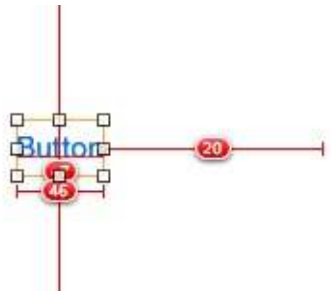
- Si arrastramos en sentido horizontal, podemos (entre otros) centrar verticalmente (aunque suene un poco a contrasentido). Y al contrario si arrastramos en vertical.
- Las restricciones de espaciado serán hacia el borde que hayamos arrastrado.

Restricciones insuficientes o contradictorias

Generalmente cuando comenzamos a añadir restricciones, las líneas que las representan aparecen en color naranja en lugar de azul. Esto sucede porque todavía **las restricciones son insuficientes** para determinar unívocamente las coordenadas del *frame* del componente. Por ejemplo si acabamos de crear un botón y lo centramos verticalmente, lo hemos "fijado" en el eje de las `x` pero no así en el de las `y`. Además se muestra un contorno dibujado en línea punteada que indica dónde calcula Xcode que acabará posicionándose el componente con las restricciones actuales (y que muy probablemente no sea donde nosotros queremos).

Otro problema típico es **mover el elemento una vez se ha establecido la restricción,** de modo que no ocupa la posición que esta indica. Las líneas de restricción también aparecerán en naranja, y el número que indica su tamaño tendrá un símbolo `+` o `-` para indicar el desplazamiento.

Cuando **las restricciones son contradictorias**, las líneas que las representan aparecen en color rojo. Por ejemplo en la siguiente figura hemos intentado especificar un espaciado de 20 puntos con el margen derecho y simultáneamente que esté centrado en horizontal. Claramente esto es imposible, y así lo indica Xcode.



Cuando hay problemas con las restricciones estos se muestran también en el `Document outline` del storyboard. En el ángulo superior derecho del `Document outline` aparece una pequeña flecha roja indicando que hay problemas, y si la pulsamos aparecerá la lista de restricciones contradictorias e insuficientes.

Podemos intentar resolver estos problemas de forma automática. Para eso está el botón `Resolve autolayout issues` de la barra de botones de autolayout (recordar que está en la esquina inferior derecha del *storyboard*). Hay varias posibilidades:

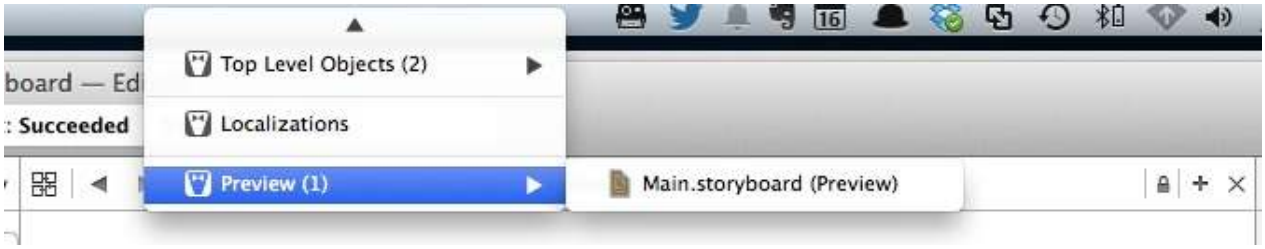
- `Update frames`: queremos recalcular las posiciones y dimensiones de los *frames* usando las restricciones actuales. Si hemos movido los elementos con el ratón, volverán a "su posición".
- `Update restrictions`: si hemos movido los elementos, Xcode intentará recalcular las restricciones para que se correspondan con la posición actual.
- `Add missing constraints`: basándose en la posición actual de los elementos, Xcode intentará inferir y añadir las restricciones adecuadas para que el *layout* deje de ser ambiguo.
- `Reset to suggested constraints`: el equivalente a eliminar todas las restricciones (`Clear constraints`) y luego seleccionar `Add missing constraints`.

Previsualizar el efecto de las restricciones

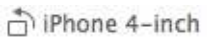
Aunque podemos visualizar el efecto de las restricciones ejecutando la aplicación en el simulador, es un proceso un poco tedioso, y más si queremos comprobar el aspecto en distintos dispositivos con distinto tamaño de pantalla. Tendríamos que ejecutar el simulador para cada uno de ellos. Desde Xcode 5 existe la posibilidad de previsualizar el aspecto de la interfaz (incluyendo por supuesto el autolayout).

> En la versión 5 de Xcode había un botón al lado de la barra de autolayout para la previsualización, que en la versión 6 ha desaparecido.

Para previsualizar la interfaz, desde el editor del *storyboard* seleccionamos el `Assistant editor` (icono  de la barra de herramientas). Como siempre sucede con este tipo de editor, el área principal se dividirá en 2. Si en la parte derecha no aparece la *preview*, la podemos seleccionar manualmente con los iconos de su zona superior



En la *preview* se ve el aspecto que va a tener la interfaz en un determinado hardware. En la parte inferior pone el nombre, por ejemplo “iPhone 4-inch”. Si pasamos el ratón por el nombre aparecerá a su izquierda un botón que sirve para rotar la pantalla.



En la parte inferior izquierda de la ventana de *preview* hay un símbolo `+` que sirve para añadir otros modelos de dispositivo a la previsualización.

Restricciones sobre el tamaño

Aunque hemos dicho que autolayout calcula el *frame* de cada componente, hasta ahora hemos ignorado el tamaño de los mismos. Centrar en horizontal y vertical elimina la ambigüedad en cuanto a en qué coordenadas “anclar” el frame pero ¿qué hay de su ancho y alto?.

Para muchos componentes (`UILabel` , `UIButton`) no es necesario especificar un tamaño ya que lo tienen por defecto (el llamado “tamaño intrínseco” en el argot de autolayout). En el API la propiedad correspondiente es `intrinsicContentSize` . Lo más habitual es que sea el tamaño del texto que contienen.

No obstante, también podemos poner restricciones sobre el tamaño. Podemos fijar el ancho y/o el alto o el *aspect ratio*. Estas son restricciones del tipo `pin` y por tanto las podemos encontrar donde encontramos las de espaciado entre componentes (en el menú principal o en la barra de botones de autolayout). Si usamos `ctrl-arrastrar` con el ratón bastará con que arrastremos sin salirnos del componente (al arrastrar en horizontal se nos dará la posibilidad de finar el ancho y lo mismo con el alto si arrastramos en vertical).

Si especificamos el tamaño mediante una restricción podemos forzar a que el contenido del botón tenga que “cortarse” porque no cabe, o bien que tenga que añadirse un *padding* al sobrar espacio.

Más sobre las restricciones

Formulación completa de una restricción

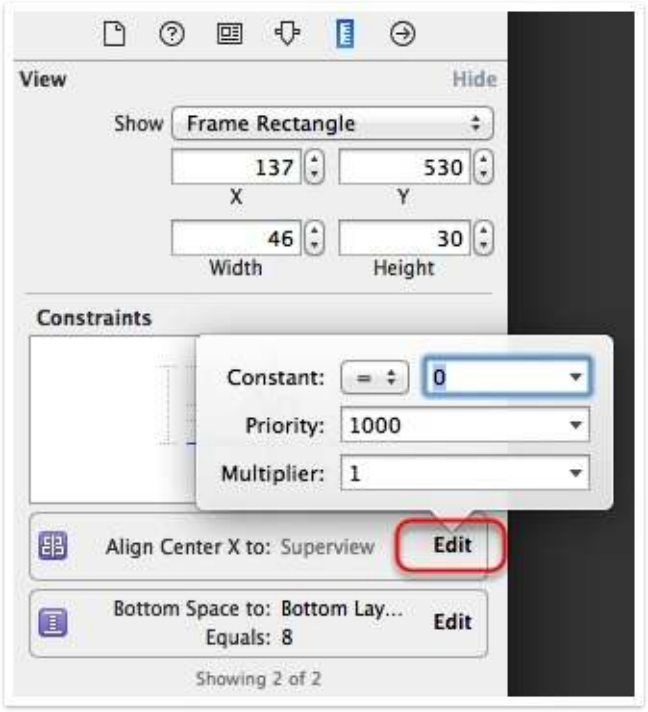
Internamente, cada restricción se formula como una ecuación lineal en la que:

```
item1.atributo1 = multiplicador * item2.atributo2 + cte
```

Algunas restricciones no son ecuaciones sino *inecuaciones*, sustituyendo el símbolo `=` por `<=` o `>=`.

> Es decir, desde el punto de vista formal, lo que hace autolayout es resolver un sistema de ecuaciones lineales.

Estas propiedades podemos verlas en el `Size inspector` (parte derecha de la pantalla, icono de la regla ). Si seleccionamos un componente de UI aparecerán aquí todas sus restricciones, que podemos editar pulsando en **Edit**. Por ejemplo, aquí vemos las restricciones de un botón centrado en el eje de las X y con un espaciado estándar (8 puntos) con respecto a la guía inferior.



Podemos observar en la figura las propiedades de la restricción, que se corresponden directamente con los coeficientes del lado derecho de la ecuación lineal (el multiplicador y la constante). Además aparece una *prioridad*, que explicaremos en el siguiente apartado. Haciendo clic en el desplegable con el símbolo `=` podemos cambiar la ecuación por una inecuación.

En nuestro ejemplo la constante es 0 y el multiplicador 1 porque queremos centrar el componente en el contenedor, es decir

```
contenedor.centerX = componente.centerX
```

Podemos por ejemplo cambiar la constante por 50, con lo que conseguiremos que el componente esté desplazado 50 puntos a la izquierda de la posición de “centrado en X”.

Si en lugar de seleccionar el componente GUI seleccionamos directamente una restricción y nos vamos al `Size inspector` podremos editar directamente las propiedades de la restricción, incluyendo también los propios atributos.

Prioridades

Cada restricción tiene asignada una **prioridad**, que es un valor numérico que especifica su “importancia” (a mayor valor, mayor prioridad). El valor por defecto es 1000, que significa que el sistema entiende que la restricción **debe cumplirse**. Valores menores que 1000 indican que el sistema intentará cumplir la restricción pero que es posible que no lo haga si hay restricciones contradictorias de mayor prioridad.

Podemos cambiar/ver la prioridad actual de la misma forma que podemos cambiar/ver el resto de propiedades de la restricción (ver apartado anterior).

Además de las restricciones, también los componentes GUI tienen dos valores de prioridad, relativos al tamaño:

- *Compression resistance*: indica la prioridad que para el componente tiene mostrar completo su contenido (resistiéndose por tanto a ser comprimido, y de ahí el nombre. Por defecto los componentes tienen este valor alto (aunque menos que 1000, en Xcode 6 está fijado a 750). Si una regla con prioridad por defecto conlleva a que el contenido del botón no se vea completo ganará la regla, pero no será así si su prioridad es menor que 750.
- *Content hugging*: indica la prioridad que para el componente tiene evitar el *padding*. Por defecto tiene un valor bajo, indicando que si hay reglas que lleven a aumentar el padding se tomarán en cuenta salvo que tengan prioridad muy baja.

Formular restricciones usando código

En lugar de usar el editor visual del Interface Builder podemos especificar las restricciones en el código fuente. Esto puede resultar interesante en diversas situaciones: a veces los elementos de la interfaz se crean dinámicamente y por tanto no se puede especificar el *layout* en Xcode. Otras veces puede ser que aunque los elementos del interfaz no cambien sí queramos que cambien dinámicamente las restricciones para conseguir distintos efectos de *layout*.

Hay dos formas de hacerlo: directamente con el API de autolayout o con un mayor nivel de abstracción usando el llamado “Visual Format Language”. Si podemos elegir, la mejor forma es la segunda, ya que es mucho más intuitivo especificar las restricciones y entenderlas leyendo luego el código.

El API básico de autolayout

Cada restricción es un objeto de la clase `NSLayoutConstraint`. Para crearla se usa el método `constraintWithItem:...` que, como vamos a ver, especifica directamente parámetro por parámetro cada una de las propiedades de la restricción. Por ejemplo, supongamos que queremos centrar un componente (que tenemos en la variable `button`) en su contenedor (variable `superview`) en el eje de las X. La restricción sería algo como

```
superview.centerX = 1*button.centerX+0
```

Donde se ha explicitado la constante y el multiplicador para ver más clara la correspondencia directa con el código, donde se haría como:

```
NSLayoutConstraint *constraint = [NSLayoutConstraint
 constraintWithItem:button
 attribute:NSLayoutAttributeCenterX
 relatedBy:NSLayoutRelationEqual
 toItem:superview
 attribute:NSLayoutAttributeCenterX
 multiplier:1.0
 constant:0.0]
```

Una vez creada la restricción para que tenga efecto hay que añadirla a la vista con `addConstraint`

```
[self.miBoton.superview addConstraint:constraint];
```

Como vemos, hemos añadido la restricción al contenedor del botón. Como norma general, si son vistas “madre/hija” la añadiremos a la “madre”, y en otro caso *al ancestro común más cercano de ambas vistas*. Por ejemplo si fuera una relación entre dos botones dentro del mismo contenedor la añadiríamos al contenedor.

Visual Format Language

La conversión de ecuación matemática a llamada del API es bastante directa, pero tiene el problema de que no es fácil y rápido deducir intuitivamente la restricción leyendo el código. Es mucho más intuitivo leer “el componente debe estar centrado en el eje X pero desplazado 10 pixels a la izquierda” que leer `superview.centroX = componente.centroX + 10`.

La descripción formal pero a la vez intuitiva de un conjunto de restricciones se puede hacer con una ingeniosa “representación en modo texto” de la representación gráfica de las restricciones llamada *Visual Format Language*. Dicho formato permite representar un conjunto de restricciones con una cadena de caracteres. La representación usa símbolos “semi-gráficos”, un poco al estilo del ASCII-ART (salvando las distancias). Así, por ejemplo si queremos especificar que entre dos componentes debe haber una separación estándar (8 pixels) usaríamos la cadena:

```
[boton1]-[boton2]
```

Donde los corchetes indican un componente, y el ``-``` indica la separación estándar. La cadena se parece razonablemente a la representación gráfica que podríamos ver en Xcode de la misma restricción.

Hay que indicar que `boton1` y `boton2` no son exactamente nombres de variables a diferencia de cuando usamos el API de `constraintWithItem:...`, sino etiquetas arbitrarias.

La llamada al API para crear esta restricción usando el *visual format language* sería algo como:

```
[NSLayoutConstraint constraintsWithVisualFormat:
   @“[cancelButton]-[acceptButton]”
   options: NSLayoutFormatDirectionLeadingToTrailing |
         NSLayoutFormatAlignAllCenterY
   metrics:nil
   views:viewsDictionary];
```

donde:

- el primer parámetro es la cadena de formato, como un `NSString*`
- `options` es una máscara de bits formada a partir de enumerados describiendo la dirección y la alineación de los componentes
- `metrics` se usa si hay constantes en la restricción (no es el caso del ejemplo). Es un diccionario en el que las claves son los nombres de las constantes y los valores son los de las constantes.
- el último, `views`, es un diccionario donde las claves son los nombres de componentes en la cadena y los valores son las variables correspondientes a las vistas. Habitualmente se usarán los mismos nombres, para simplificar, en cuyo caso podemos crear el diccionario automáticamente con `NSDictionaryOfVariableBindings`, al que le pasamos un número variable de argumentos con las variables de las vistas, por ejemplo:

```
UIButton *cancelButton = ...;
UIButton *acceptButton = ...;
NSDictionary *views = NSDictionaryOfVariableBindings(cancelButton,
                  acceptButton);
```

Algunos ejemplos adicionales de cadenas de formato:

- `[boton1]-20-[boton2]` separación de 20 puntos
- `[boton1(50)]-20-[boton2(>=50)]` entre paréntesis especificamos el ancho del botón, nótese que se pueden poner desigualdades
- `[boton1]-20@800-[boton2]` las prioridades se ponen con la `@`
- `[boton1]-20-[boton2(==boton1)]` el botón 2 debe ser del mismo tamaño que el 1.
- `V:[topField]-10-[bottomField]` con la `V` especificamos que es un *layout* en vertical, los dos campos estarán uno encima del otro separados por una distancia de 10 puntos.
- `|-[find]-[findNext]-[findField(>=20)]-|` una línea completa de *layout*, donde las barras verticales representan los bordes del contenedor.

Se recomienda consultar la documentación de Apple para más información sobre la sintaxis y ejemplos adicionales.

> En el diseño del formato, se ha preferido la claridad y el paralelismo con la representación gráfica a la expresividad. Como resultado, ciertas restricciones no son expresables. Por ejemplo no se puede especificar que el ancho de un botón sea el doble que el de otro.

Referencias

- Videos de las sesiones de la Apple WWDC 2012 (requieren un id. de Apple)
  - Introduction to Auto Layout for iOS and OS X
  - Best Practices for Mastering Auto Layout
- Tutorial de Ray Wenderlich
- Tutorial Avanzado de Objc.io

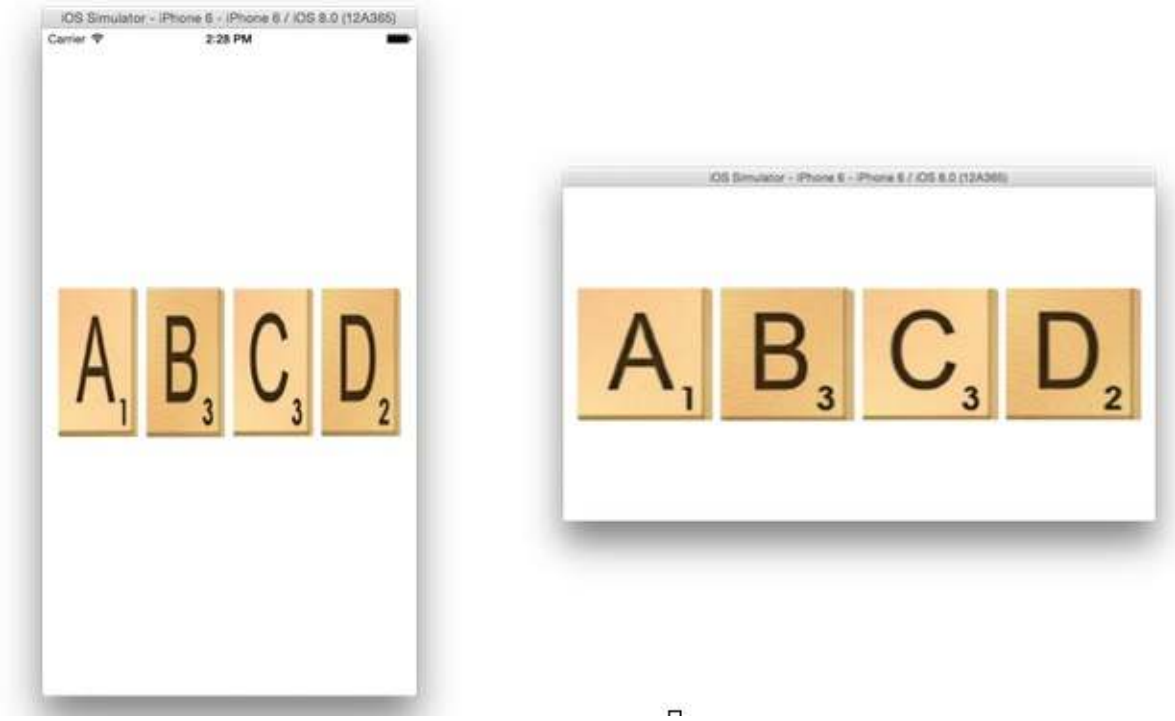
Ejercicios de autolayout

Aplicación “Pioneras”

Añade *autolayout* a la aplicación que hicistes ayer en la sesión 1 (“Pioneras”). El ojetivo fundamental es intentar que se vea bien en modo *portrait* (vertical) independientemente del tipo concreto de iPhone (4, 5, 6…). Ten en cuenta que del iPhone 4 al 5 hay un cambio en la altura de la pantalla y del 5 a 6 un cambio en altura y anchura.

layout tipo “Acordeón”

En las plantillas de la sesión hay unas cuantas imágenes de fichas de *scrabble*.El objetivo es hacer que aparezcan centradas en vertical, una al lado de la otra. Entre cada imagen debe haber una distancia fija y todas las imágenes deben tener el mismo ancho. Si el ancho de la pantalla cambia, debería cambiar automáticamente el ancho de las imágenes para que la separación siga siendo la misma.



- Crea un proyecto llamado `Autolayout`
- Añade las imágenes al `images.xcassets` como es habitual.
- Usa la interfaz visual de Xcode para poner las restricciones que consideres necesarias
- comprueba que funciona independientemente del modelo de iPhone y de la orientación (sea *portrait* o *landscape*).

layout por código

En otra pantalla del *storyboard* inserta la imagen de la `A` del *scrabble* e intenta usando objective-c (sin el editor visual) que aparezca centrada en vertical y que el ancho se adapte automáticamente, dejando una distancia fija a los lados. Es decir, como el “acordeón” del ejercicio anterior, pero solo con un objeto, para simplificar.

- Pon la pantalla como controller inicial para que empiece a ejecutarse desde aquí, o crea un segue para llegar hasta ella
- crea un nuevo controller para meter el código (una `Cocoa Touch class` que herede de `UIViewController`
- asocia el controller a esta pantalla, con el *identity inspector*

Toma como punto de partida el siguiente código, al que debes añadir el tuyo

```
//En el viewDidLoad del controller
UIImageView * imgView = [[UIImageView alloc] initWithImage:
            [UIImage imageNamed:@"a"]];
[self.view addSubview:imgView];
//no usar el método antiguo de layout (autoresizing masks)
//porque usar este junto con autolayout lleva al desastre
[imgView setTranslatesAutoresizingMaskIntoConstraints:NO];
```

Sesión 4: tablas

Introducción

Las vistas de tabla (`UITableView`) se encargan de mostrar, gestionar y hacer *scrolling* de una tabla de elementos de una sola columna. Cada una de las filas se modela con un `UITableViewCell` .

> Si necesitamos más de una columna podemos usar `UICollectionView` , que veremos en la parte “avanzada” de la asignatura.

El aspecto de las tablas es enormemente configurable, lo que hace que aparezcan múltiples veces en sus distintas “encarnaciones” en muchas aplicaciones iOS, por ejemplo, en las aplicaciones de Mail, Ajustes, Reloj…

Las tablas pueden ser *simples* (`UITableViewStylePlain`) o *agrupadas* (`UITableViewStyleGrouped`)

Hay varios estilos predefinidos para las filas, que nos permiten mostrar diversos elementos: título, subtítulo, icono a la izquierda, … También podemos crear nuestros propios estilos de celda bien por código o bien gráficamente en el *interface builder*.

Creación de vistas de tabla

Una vista de tabla interactúa básicamente con tres objetos (aunque podemos implementar todas las funcionalidades en una única clase, como se suele hacer en los casos más simples)

En primer lugar, el *view controller*. Ya hemos visto el papel que hace en las aplicaciones.

En segundo lugar, el *data source*: las vistas de tabla solo almacenan directamente los datos de las filas actualmente visibles en pantalla. El resto se los piden a un objeto que debe ser conforme al protocolo `UITableViewDataSource` . Este es obligatorio, no podemos crear una tabla sin él.

Y finalmente el *delegate*: para gestionar algunos eventos de manipulación de la tabla (como la edición, el borrado, o el mover una fila) y controlar algunos aspectos de la apariencia de las celdas, se usa el `UITableViewDelegate`

Es muy habitual que el *controller*, el *delegate* y el *data source* sean el mismo objeto.

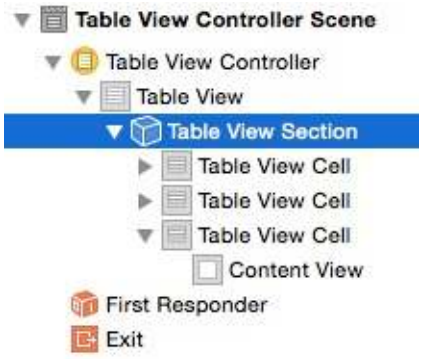
Si usamos una vista de tabla dibujada en el *storyboard* podemos conectarla con los dos “colaboradores” gráficamente mediante el “Connections Inspector” del panel “Utilities”

Tablas estáticas

En algunos casos conocemos de partida los elementos que queremos dibujar en la tabla. Ejemplo típico de esto es la aplicación de *Ajustes*, en la que las opciones están colocadas en una tabla simplemente para que estén más organizadas y tengan un formato atractivo. Esto lo podemos conseguir con una *tabla estática*.

Para crear una pantalla con una tabla estática arrastramos un `Table View Controller` al *storyboard*. Es un *controller* asociado a una vista de tabla que ocupa toda la pantalla del dispositivo. Por defecto usa una tabla dinámica, pero podemos cambiarlo seleccionando la tabla en el `Attributes inspector` y seleccionando `Static Cells` en la primera propiedad, `Content` .

Podemos añadir secciones a la tabla y cambiar el número de celdas en cada sección. Para poder cambiar el número de celdas hay que tener seleccionada la sección deseada, lo que a veces es difícil con el ratón, por lo que podemos usar los nodos del `Document outline` :



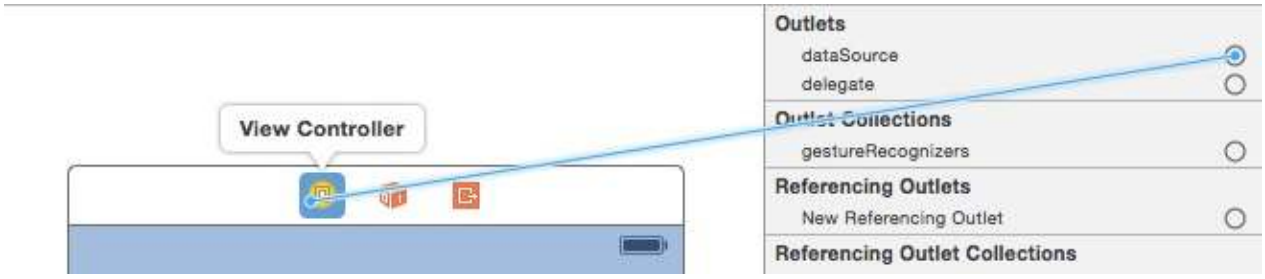
> Podemos aumentar el número de celdas de modo que no quepan en la pantalla. Para desplazarnos por la tabla podemos seleccionarla y hacer *scroll* con la rueda del ratón

Dibujar tablas dinámicas

El encargado de decirle a la vista de tabla qué contenido debe dibujar es el *datasource*. Este puede ser el objeto que nosotros queramos con tal de que implemente el *protocol* `UITableViewDataSource`. Habitualmente será el *controller* de la pantalla en la que está la tabla.

Conectar la tabla y el *datasource*

Podemos conectar gráficamente la tabla con el *datasource* mediante el `Connections inspector` (el icono de la flecha - el último - en el área de `Utilities`). Si tenemos seleccionada la tabla aparecerán las propiedades que podemos conectar, entre ellas el `datasource`. Arrastramos (no hace falta `Ctrl`) desde el círculo que representa al `datasource` hasta el icono del *view controller* (el primero de los tres que aparecen en la parte superior de cada pantalla del *storyboard*)



Implementar los métodos del protocolo `UITableViewDataSource`

Como mínimo el objeto que actúe de *datasource* debe implementar dos métodos:

- `tableView:numberOfRowsInSection:` que debe devolver el número de filas que tiene una determinada sección (de la única si solo hay una)

```
- (NSInteger)tableView:(UITableView *)tableView numberOfRowsInSection:(NSInteger)section {
    //supongamos que la propiedad "datos" es un array con los datos de la tabla
    return [self.datos count];
}
```

- `tableView:cellForRowAtIndexPath:` debe devolver la celda correspondiente a un determinado número de fila y sección. Este merece una discusión más detallada que el anterior

Lo más simple sería construir un nuevo objeto `UITableViewCell` por cada fila:

```
- (UITableViewCell *) tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    //Construimos la celda y le damos un estilo de los predefinidos
    UITableViewCell *celda = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"UnaCelda"];
    //Instanciamos el texto de la celda
    celda.textLabel.text = self.datos[indexPath.row];
    return celda;
}
```

Aclaraciones:

- un objeto `indexPath` especifica el número de fila (`row`) y de sección (`section`) con las propiedades del mismo nombre.
- En un momento veremos qué es el `reuseIdentifier` del método inicializador de la celda. Por ahora simplemente le asignamos un `NSString` arbitrario

> Nota: el código anterior no se suele implementar tal cual en la realidad. Sería muy ineficiente por las razones que veremos a continuación.

- Crear una nueva celda por cada fila es muy ineficiente dado que una tabla puede tener cientos o miles de ellas. Por eso se suele usar un truco ingenioso: se crea un pequeño número de celdas (las que se ven simultáneamente en pantalla) y luego se reutilizan conforme se va haciendo *scroll* por la tabla, rellenándolas con los nuevos datos. iOS ofrece soporte para esta reutilización manteniendo un *pool* de celdas que podemos reutilizar para rellenar con nuevos datos.
  - Podemos sacar una celda del *pool* con `dequeueReusableCellWithIdentifier:` .
  - El identificador sirve para etiquetar el “tipo” de celda. En casos más complicados podríamos tener varios *pool* con distintos tipos de celda para reutilizar. En el ejemplo usaremos un único identificador (es arbitrario y lo elige el desarrollador)

```
- (UITableViewCell *) tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    //Solicitamos una celda del "tipo" desedo al "pool"
    //Esto es mucho más rápido que crearlas desde cero
    UITableViewCell *celda = [tableView dequeueReusableCellWithIdentifier:@"UnaCelda"];
    //si nos ha devuelto nil es que no habían celdas disponibles.
    //Tendremos que crear una, como hacíamos antes
    if (celda == nil) {
        celda = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault reuseIdentifier:@"UnaCelda"];
    }
    //Igual que antes rellenamos los datos y devolvemos la celda
    celda.textLabel.text = self.datos[indexPath.row];
    return celda;
}
```

- Si cambiamos el conjunto de datos a mostrar en la tabla, para que los cambios se reflejen en la pantalla deberemos llamar al método `reloadData` de la vista de tabla.

Gestión de tablas

En las tablas dinámicas podemos por supuesto insertar y eliminar celdas. También podemos seleccionarlas haciendo *tap* sobre ellas. El encargado de procesar todas estas tareas es el *delegate* de la tabla

Seleccionar celdas

Cuando se selecciona una celda, el *delegate* recibe el mensaje `tableView:didSelectRowAtIndexPath:`

```
- (void) tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *celda = [tableView cellForRowAtIndexPath:indexPath];
    //Colocamos un "checkmark" en la celda o lo quitamos si ya estaba
    if (celda.accessoryType==UITableViewCellAccessoryNone)
        celda.accessoryType = UITableViewCellAccessoryCheckmark;
    else
        celda.accessoryType = UITableViewCellAccessoryNone;
    //Hacemos que la celda se deseleccione visualmente
    [tableView deselectRowAtIndexPath:indexPath animated:YES];
}
```

Insertar y eliminar filas

iOS nos ofrece de forma automática el “modo edición” en el que en cada celda aparece una señal de “prohibido” para poder borrarla (pulsando sobre la señal y luego sobre el botón “Borrar” que aparece). Podemos activar este modo con el método `setEditing`:

```
//"miTabla" es un outlet a la vista de tabla
[self.miTabla setEditing:YES animated:YES];
```

Aunque el modo edición es automático, el borrado efectivo de las celdas y de su contenido lo tenemos que hacer nosotros, al igual que la inserción. Podríamos usar un código similar al siguiente:

```
 (void) tableView:(UITableView *)tableView
           commitEditingStyle:(UITableViewCellEditingStyle)editingStyle
           forRowAtIndexPath:(NSIndexPath *)indexPath {
   //si el usuario ha pulsado sobre borrar
   if (editingStyle == UITableViewCellEditingStyleDelete) {
       //Nos cargamos el objeto de la lista de valores
       //IMPORTANTE: hay que borrar primero del modelo antes que
       //la celda de la tabla
       [self.datos removeObjectAtIndex:indexPath.row];
       //lo borramos visualmente de la tabla
       [tableView deleteRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                 withRowAnimation:UITableViewRowAnimationFade];
   }
  //si está insertando una nueva fila
   else {
       NSString *nuevoTexto = self.miCampo.text;
       [self.datos insertObject:nuevoTexto atIndex:indexPath.row];
       [tableView insertRowsAtIndexPaths:[NSArray arrayWithObject:indexPath]
                   withRowAnimation:UITableViewRowAnimationAutomatic];
   }
}
```

Podemos controlar cómo queremos que aparezca cada celda en el “modo edición”

```
- (UITableViewCellEditingStyle) tableView:(UITableView *)tableView
              editingStyleForRowAtIndexPath:(NSIndexPath *)indexPath {
    //En la última celda hacemos que aparezca un icono de "insertar"
    if (indexPath.row==[self.datos count]-1)
        return UITableViewCellEditingStyleInsert;
    //En el resto que se puedan borrar
    else
        return UITableViewCellEditingStyleDelete;
}
```

Sesión 5: controladores contenedores

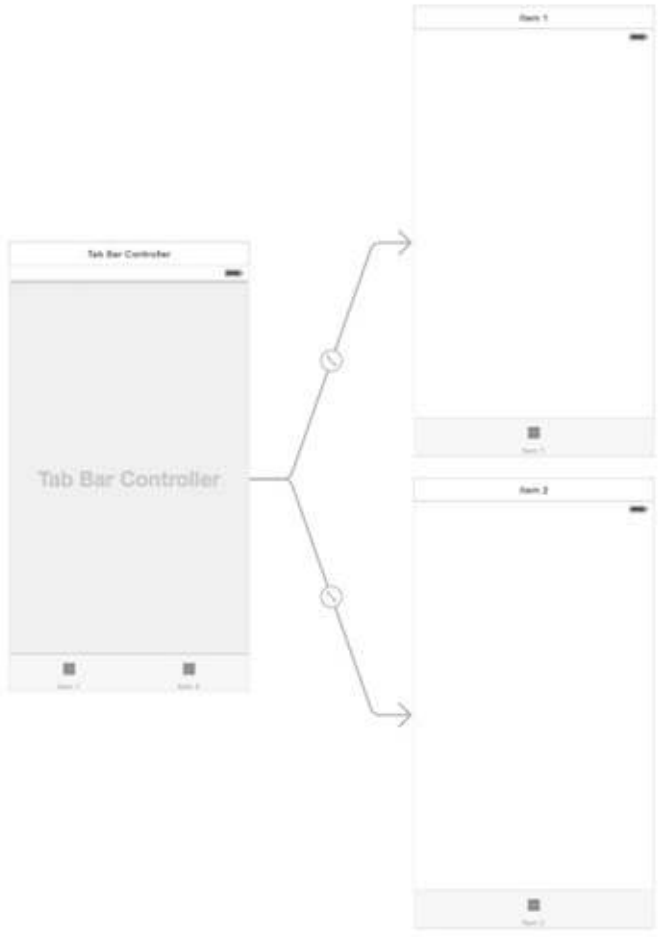
A diferencia de los otros tipos de *controllers* que hemos visto, los controladores contenedores no muestran directamente el contenido “principal” de la aplicación. El papel de estos controladores es, como su propio nombre indica, el de servir de contenedores a los controladores que muestran el contenido y permitir la navegación sencilla entre ellos.

En esta parte básica de la asignatura vamos a ver los dos controladores contenedores más típicos: el *tab bar* y el *navigation*.

Tab bar controllers

Permiten diseñar la típica aplicación dividida por “apartados” o “tabs”. Por ejemplo lo podemos ver en la aplicación de “salud” de iOS 8 (entre muchísimas otras)

Lo que tenemos es un controlador de tipo “tab bar” que gestiona la barra inferior y la navegación entre pantallas cuando pulsamos cada uno de sus iconos, pero el contenido de las pantallas lo gestionan los otros controladores. Podemos ver la estructura de forma más clara si arrastramos un *tab bar controller* al *storyboard* desde la *object library*.



Vemos que el nuevo *tab bar controller* aparece conectado a dos controladores convencionales (de la clase `UIViewController`). Automáticamente cuando pulsemos en cada icono, se saltará al controller asociado.

Podemos añadir pantallas a este *tab bar controller* sin más que crearlas en el *storyboard* y luego conectarlas. Hacemos `Ctrl+arrastrar` desde el *tab bar controller* hasta la pantalla a conectar y en el menú contextual elegimos el tipo de *segue* llamado `view controllers`.

Otra forma de crear un *tab bar controller* relacionado con una pantalla que ya tengamos creada es seleccionar la misma y en el menú de `Editor` elegir `Embed in > Tab bar controller`. Aparecerá un tab bar controller con un único icono en la barra inferior, y ya conectado a la pantalla actual.

Personalizar la barra inferior

Desde el *interface builder* podemos cambiar algunas propiedades básicas de los iconos de la barra inferior en el `attribute inspector`, como el icono, el título, la posición del título con respecto al icono, si queremos que aparezca un *badge*, etc.

> Si elegimos uno de los iconos del sistema no podremos cambiar el título ya que Apple considera que lo contrario podría inducir a confusión al usuario

Si queremos personalizar el icono poniendo nuestra propia imagen hay que tener en cuenta que los iconos de un *tab bar* son monocromáticos. El formato a usar es .png, del que el sistema examinará el canal alfa o de transparencia. Tomará como forma del icono los pixels que sean opacos ignorando su color. Podéis consultar más detalles sobre el formato y el tamaño recomendado en el apartado “Bar button icons” de las *iOS Human Interface Guidelines* de Apple.

Navigation controllers

Los *navigation controller* sirven para crear estructuras de navegación jerárquica en las que tenemos pantallas y

“subpantallas”. Estando en una de ellas podemos volver atrás una a una. El *navigation controller* se encargará de que se vaya cambiando automáticamente al controlador apropiado (el de la pantalla actual).

Nótese que cuando estamos navegando de este modo estamos usando una **pila de controladores**, en el sentido que tiene este término en estructuras de datos, ya que conforme vamos profundizando en la jerarquía se van apilando los controladores, y cuando volvemos atrás quitamos el de la parte de arriba de la pila.

Crear un controlador de navegación

Hay dos formas de crear gráficamente un controlador de navegación:

1) Seleccionamos un *controller* en el *storyboard* y elegimos la opción de menú de `Editor > Embed in > Navigation controller`. Se creará un controlador de navegación que tiene como controlador de contenido asociado al que habíamos seleccionado.

Nótese que como en el caso de los *tab bar controller* en el de navegación la pantalla está en gris indicando que el contenido no es responsabilidad de este controlador. En el controlador de contenido asociado podemos ver que la parte superior aparece en gris, indicando que esta es la *barra de navegación* que gestionará el controlador de navegación.

2) Podemos arrastrar un `Navigation controller` desde la librería de objectos hasta el *storyboard*. Por defecto nos creará el controlador de navegación junto con un controlador de tabla, estructura apropiada para comenzar con una vista maestro/detalle. No obstante si no queremos este segundo controlador podemos borrarlo y conectar el de navegación con uno nuevo con `Ctrl+Arrastrar` entre ambos y seleccionando en el menú contextual la opción de `root view controller` bajo `relationship segue`.

Añadir pantallas a la jerarquía

Para añadir un controlador a la jerarquía de navegación basta con `Ctrl+Arrastrar` entre el componente que dispararía el salto y el controlador de destino. En el menú contextual hay que elegir el *segue* de tipo *push*. Evidentemente el controlador de origen tiene que estar ya en la jerarquía de navegación para que esta operación funcione.

Personalizar la barra de navegación

Automáticamente el controlador de navegación gestionará una barra de navegación en la parte superior de la pantalla. Esta barra muestra por defecto en su parte izquierda un botón `< Back` para ir al controlador anterior.

La barra de navegación es totalmente personalizable. Lo más inmediato es mostrar un título para la pantalla actual cambiando la propiedad `title` del controlador. Podemos hacerlo por ejemplo en el método `viewWillAppear` del mismo, ya que se ejecutará antes de mostrar la vista. Al cambiar el título del controlador también cambiará automáticamente el botón ``< Back``` para reflejar el nuevo título.

> Si el título del *controller* es demasiado largo y no cabe en el botón este seguirá con el título por defecto (`Back`)

Podemos cambiar completamente la barra de navegación. Es accesible mediante la propiedad `navigationItem` del controlador, y podemos cambiar por ejemplo:

- `backBarButtonItem` : el botón, de tipo `UIBarButtonItem` , que se utilizará para volver atrás a este controlador
- `titleView` : el componente con el título para el controlador, cuyo texto como hemos visto podemos cambiar con la propiedad `title` del controlador.
- `rightBarButtonItem` : por defecto `nil` . Típicamente se usa para añadir un botón para editar el contenido de la pantalla actual, pero podemos colocar lo que queramos.