Titanium Test Scores

UI/UX Design	30%
Tables	40%
User Input	40%
Orientation	40%
Modules	40%
Scroll Views	60%

******* Study Notes *******

User Interface Deep Dives

Tab recommendations
Users expect each tab to be focused on a limited and related set of functionality. Tabs are expected to be related to each other, within the overall purpose of your app. However, they are not expected to be in any sort of hierarchical relationship. Tabs are peers, or siblings, of each other; not children of one another.

Units
Placement and dimensions of UI elements are specified using a numeric value plus an implicit or explicit unit of measurement. If you don't specify a unit of measurement, the system unit is assumed. You can also set a default unit of measurement to use in your app by setting a tiapp.xml property.

First, a couple of definitions we'll use in the rest of this guide:

- dip : Density-independent pixels. A measurement which is translated natively to a corresponding pixel measure using a scale factor based on a platform-specific "default" density, and the device's physical density.
- System unit : A platform-dependent unit which is the default for how the system presents its view information to the native layout system. On Android this is pixels; on iOS it is dip.

Supported units are:

- Absolute measurements
  - px : pixels
  - mm : millimeters
  - cm : centimeters
  - in : inches
  - dp/dip : Density-independent pixels (we sometimes call these "points")
    - Android : actual pixels = dip * (screen density) / 160
    - iOS : actual pixels = dip * (screen density) / 163 (effectively 1dip=1px on standard, 1dip=2px on retina)
    - Mobile Web: actual pixels = dip * (screen density) / 96 (effectively 1dip=1px because most browsers scale pages to 96dpi to make them consistent with desktops).
- Relative measurements
  - % : Percentage of the size of the parent.
    - For x-axis values (width, left, right, center.x) this is relative to the parent's width
    - For y-axis values (height, top, bottom, center.y) this is relative to the parent's height.

The coordinates grid
Titanium uses a grid coordinate system for layout. Grid locations are based on the system unit (platform-dependent unit). This means that by default on iOS, elements are positioned on a density-independent grid and on Android on a density-dependent grid. The net result is that on iOS, elements are positioned in visually the same locations regardless of the actual density of the screen. On Android, elements are positioned at the same absolute pixel locations and might lay out differently depending on the device.

- iPhone with either original or retina display is based on a 320 x 480 dip grid.
- iPad is based on a 1024 x 768 dip grid.
- Android device screen sizes vary. Considering these emulator examples:
  - HVGA emulator is 320 x 480 px
  - WVGA800 emulator is 480 x 800 px
  - WVGA854 emulator is 480 x 854 px

Remember that you can specify `dp` or `dip` units on Android (and even set an app-level default in tiapp.xml) to achieve the same density-independent grid as offered by default on iOS.

Positioning and dimensions of elements
Elements in Titanium are positioned relative to their parent container, such as the window or a view. Depending on the positioning properties you use, the reference point will be either the parent's top/left or bottom/right corner. We call this the "view hierarchy." Options include:

- `top` and `left` properties, which specify the grid position of the element's top/left corner relative to the parent's top/left corner.
- `bottom` and `right` properties, which specify the grid position of the element's bottom/right corner relative to the parent's bottom/right corner.
- `center` property, which species the position of the element's center point relative to the parent's top/left corner.

In the following example, the red view is positioned at the 20,20 point relative to the window's top/left corner.

The yellow view's bottom/right corner is 100 points/pixels from the bottom/right corner of the display.

The blue view's center is at 160,240 and given its width of 50, this means its top-left corner would be at 110,190.

The green view has a sufficiently negative `top` value given its width that it is positioned off the top of the screen.

Layout modes
Titanium Windows and Views can employ one of three layout modes by setting its `layout` property to one of the following values:

- `absolute` - This is the default mode that we have discussed to this point. You specify point coordinates on a grid relative to the parent container's top/left or bottom/right corner.
- `vertical` - This layout mode stacks child views vertically. The child's `top` property becomes an offset value. It describes the number of units from its previous sibling's bottom edge where the view will be positioned.
- `horizontal` - This layout mode lines up child views horizontally. The child's `left` property, similar to `vertical`, becomes an offset. This time, it's the position from

the previous sibling's right edge.

zIndex & default stacking

You can position elements atop one another. By default, as you add views to a parent container, they will overlay any views you previously added (assuming their boundaries overlap). You can control the stacking order by either changing the order you add elements to the container (not always convenient) or by setting the `zIndex` property. As with HTML elements, `zIndex` accepts an integer value of zero or greater. The higher the `zIndex` value, the closer to the top of the stack a view will become.

1. TableViews

```
Ti.UI.createTableView()
```

*** Add data to table**

```
// create an array of anonymous objects
var tbl_data = [
     {title:'Row 1'},
     {title:'Row 2'},
     {title:'Row 3'}
];
// now assign that array to the table's data property to add those objects as rows
var table = Titanium.UI.createTableView({
     data:tbl_data
});
// alternatively, you could do
table.setData(tbl_data);

var row = Titanium.UI.createTableViewRow({
    title: 'Row 1'
    /* other properties */
});
table.appendRow(row);
// with an explicit object, you can call methods such as
// var imgCapture = row.toImage();
```

*** Remove data**

```
table.setData([]);
// or
table.data = [];
```

For performance use setData() with an array of data or set the data property to an array.

Row properties

Now that we've seen how to create tables and rows, let's learn a bit more about the built-in row properties. TableViewRow objects have various useful properties that you can use to add style and functionality to your tables.

- `className` – set this property equal to an arbitrary string to optimize rendering performance. On both iOS and Android, setting this property enables the operating system to reuse table rows that are scrolled out of view to speed up the rendering of newly-visible rows. On iOS, the string you supply is used to specify the reuse-identifier string (setdequeueReusableCellWithIdentifier); on Android, it is used within a custom object reuse method within Titanium.
- `leftImage` – set this property equal to an image URL (local or remote) to display that image to the left of the row's title
- `rightImage` – set this property equal to an image URL (local or remote) to display that image to the right of the row's title
- `backgroundImage` – set this property equal to an image URL (local or remote) to display that image in the background of the row
- `backgroundColor` – set this property to a color string to set the row's background color
- `hasChild` – indicates sub-table or additional rows (most commonly used on iOS with the NavigationGroup control)
- `hasDetail` – indicates a detail view or alert will appear when row is tapped (not supported on Android)
- `hasCheck` – an on/off or yes/no indicator

2. Scrolling Views

ScrollView properties

There are a few interesting properties on the ScrollView. These include:

Property	Description
zoomScale, minZoomScale, maxZoomScale	You can control zooming of the content within the ScrollView with these properties. Each accepts a numeric value between 0 and 1.
horizontalBounce, verticalBounce	(iOS only) These Boolean values control whether the ScrollView displays that "bounce" effect when the user has reached the end of the scrolling content.
showHorizontalScrollIndicator, showVerticalScrollIndicator	These Boolean values control whether the scroll indicator (scrollbar-like gizmo) is displayed.
scrollType	On Android, you can set the ScrollView to either "vertical" or "horizontal" but not both.
canCancelEvents	On iOS, you can set this value to `true` (default) so that events are handled by the ScrollView rather than the views it contains.

ScrollableView properties

There are a few interesting properties on the ScrollView. These include:

Property	Description
showPagingControl	Boolean, set to `false` (default) to hide the paging control (the dots that show which page you're viewing)
pagingControlColor	Set the background color for the paging control; you can't control the color of the dots.

pagingControlHeight	Set the height of the paging control area.
currentPage	This property accepts an index number of the view to display (zero-based, so `currentPage=2` would show the third view within the ScrollableView)
cacheSize	This iOS-only property accepts an integer value to control the number of views pre-rendered. See the API docs for considerations when using this property.

ScrollableView methods

Method	Description
scrollToView()	Accepts an integer or object reference of the sub-view to scroll into view within the ScrollableView.
addView()	Adds a view to the ScrollableView, as shown in the code above.
removeView()	Removes a view from the ScrollableView, as shown in the code above.

3. Orientation

With Android, you can set the UI orientation to any of four possibilities: **portrait upright, landscape right, portrait upside-down, and landscape left.**

But, when you request the current orientation, you'll get one of two values: **portrait** or **landscape**.

A phone is in **portrait** mode when its "**top**" is at 0 degrees **(hardware buttons at the bottom)** and **landscape** when the "top" is at 270 degrees.

A tablet is in **landscape** mode when its **top** is at 0 and portrait when its top is at 90 degrees. (Based on sensor degrees.)

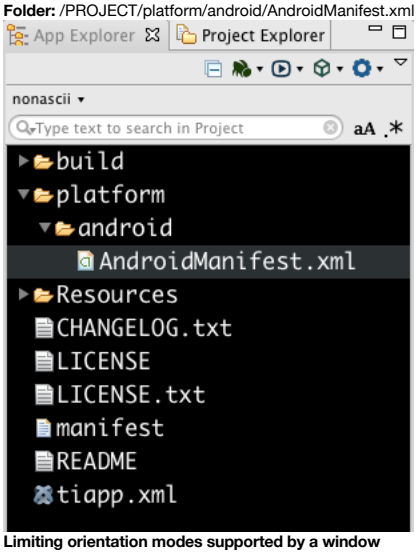
These portrait/landscape values are what you receive when you *get* the devices current orientation.

- On iPhone/iPod Touch – **don't mix orientation of windows within a single app**; so, either lock orientation for the whole app, or react to orientation changes.
- On iPhone – **don't support the portrait-upside-down orientation**, because that could leave the user with their phone upside-down when receiving a phone call.
- On iPad – you should support all orientations because that matches how people use those devices.

```
<iphone>
    <orientations device="iphone">
        <orientation>Ti.UI.PORTRAIT</orientation>
    </orientations>
    <orientations device="ipad">
        <orientation>Ti.UI.PORTRAIT</orientation>
        <orientation>Ti.UI.UPSIDE_PORTRAIT</orientation>
        <orientation>Ti.UI.LANDSCAPE_LEFT</orientation>
        <orientation>Ti.UI.LANDSCAPE_RIGHT</orientation>
    </orientations>
</iphone>

<android xmlns:android="http://schemas.android.com/apk/res/android">
  <manifest>
    <activity
    android:name="org.appcelerator.titanium.TiActivity"
    android:configChanges="keyboardHidden"
       android:screenOrientation="portrait"
    />
    <activity android:name="ti.modules.titanium.ui.TiTabActivity"
      android:configChanges="keyboardHidden"
    />
  </manifest>
</android>
```

Custom AndroidManifest.xml
Folder: /PROJECT/platform/android/AndroidManifest.xml



Limiting orientation modes supported by a window

```
var win = Ti.UI.createWindow({
     /* on Android, it needs to be a "heavyweight" window */
     fullscreen: false,
     /* This works on iOS */
     orientationModes: [
          Ti.UI.PORTRAIT,
          Ti.UI.UPSIDE_PORTRAIT
     ]
});
// but for Android, you have to set it after creation
win.orientationModes = [Ti.UI.PORTRAIT, Ti.UI.UPSIDE_PORTRAIT]
```

Reacting to orientation changes

```
Ti.Gesture.addEventListener('orientationchange',function(e) {
     // get current device orientation from
     // Titanium.Gesture.orientation

     // get orientation from event object
     // from e.orientation

     // Ti.Gesture.orientation should match e.orientation
     // but iOS and Android will report different values

     // two helper methods return a Boolean
     // e.source.isPortrait()
     // e.source.isLandscape()
});
```

4. User Input

5. Modules

Installing a module for a single project
To install a module for a single project:

1. Copy the module's zip file to the Resources directory of your Titanium project.
2. Build your project, at which point Titanium's python scripts will un-zip the module and copy the files to the appropriate directories within your project's directory hierarchy.

Alternatively, you could unzip the module's distribution file and copy the resulting directory tree to your ***Project** root* directory (the folder in which your app's tiapp.xml file is located).

Installing a module for all projects on a computer
To install a module so that it will be available to any project you create on your computer:

1. Copy the module to the root of your Titanium installation; see the table below for the location of this directory for your operating system and version.
2. Build your project, at which point Titanium's python scripts will un-zip the module and copy the files to the appropriate directories.

Operating System	Local Path
OSX (Pre-Lion)	`/Library/Application Support/Titanium`
OSX (Lion)	`~/Library/Application Support/Titanium`
Windows 7	`%ProgramData%\Titanium\mobilesdk\win32`
Windows XP	`C:\Documents and Settings\All Users\Application Data\Titanium`
Linux	`~/.titanium/mobilesdk/`

Configuring your app to use a module

Once you have installed the module to your project or system, you must configure your app to use it. This involves two steps:

- Update the application's `tiapp.xml` file.
- `require()` the module within your JavaScript code

Updating tiapp.xml
In the `tiapp.xml` file, inside the `<ti:app>` node, modify the `<modules/>` tag like this:

```
<!-- $MODULE_VERSION should be the same as "version" in the module manifest and directory number -->
<modules>
  <module version="$MODULE_VERSION">$MODULE_ID</module>
  <!-- For example, if we were adding the calc module: -->
  <module version="0.1">org.appcelerator.calc</module>
</modules>
```

Using require() to load the module in the app's code
Within your app's JavaScript files, you'll instantiate the module via the `require()` function:

```
var Module = require('$MODULE_ID');
// For example, to load the calc module:
var Calc = require('org.appcelerator.calc');
```

Step 1: Creating an Android Module

```
> titanium create --platform=android --type=module --name=calc --id=org.appcelerator.calc --android=/path/to/android-sdk
```

Step 2: Understanding the Different Parts of a Module

A number of files and directories work together to define our module and what it does. Let's take a quick look at each of them:

- **LICENSE** - The module's full license text; this will be distributed with the module to let other developers know how they can use and redistribute it.
- **build.properties** - An Ant properties file that contains paths to the Titanium SDK and Android SDK on your computer.
- **build.xml** - The main Ant build script used to build, test, and finally distribute the module.
- **manifest** - Contains the version, author, license, copyright, name, id, GUID, and platform information for the module.
- **timodule.xml** - A place to put custom activities, and general XML that will end up in the AndroidManifest.xml of apps. Read more about this file in the tiapp.xml and timodule.xml reference.
- **hooks** - A directory with scripts that will be called when a module is added/installed/removed/uninstalled from a project (this is still a work in progress).
- **documentation** - Documentation in Markdown format that is shipped with your module after being compiled to HTML.
- **assets** - Module specific assets such as images. For more information, open up the README file located in this directory.
- **lib** - Place any third party JAR dependencies here and they will be bundled up as a part of your module automatically.
- **src** - The source code for the module.
- **example** - The module example project that will be: 1) used to test the module, and 2) bundled with the module for other developers to reference.
- **platform** - This optional folder can include an "android" subdirectory, and then any of the resource directories defined in Android's Defining Resources guide (such as "res").
- **jni** - This optional folder (as of Titanium SDK 2.1.0) can include C or C++ code to compile shared libraries. You are required to have an `Application.mk` file in this directory if it is present. Modules using JNI or NDK support via shared libraries will work with both the V8 and Rhino runtimes.

Step 3: Importing a Module into Eclipse

When we ran titanium create, project settings for Eclipse were created as well. That means we can import the project very easily. In Eclipse, simply follow these steps:

- In the top level menu, Click on *__File__* > *__Import...__*
- Expand the *__General__* folder, and double click on *__Existing Project into Workspace__*
- Click on *__Browse...__* next to the *__Select root directory__* text field
- Choose your module project's folder
- You should see your module project under the *__Projects__* list:

Step 4: Titanium APIs for Module Development

For a complete list of the APIs that can be used from an Android module, look at our Android Module API.

Here is a quick primer on the APIs that every module uses:

Modules and Proxies

- A module is a class that provides an API point with a particular ID. That ID can then be used to require('the.particular.id') the module from JavaScript.
- A proxy is a class that describes how JavaScript can use it.
- Modules are proxies, plus a couple additional differences to let them be require'd and to let them receive application level events and lifecycle notices.
- Proxies can expose methods, properties, and constants to JavaScript. Each of these can be a primitive type, or a proxy.

Modules

- Must extend KrollModule and have the @Kroll.module annotation.
- Can have a parent module: Titanium.UI and Titanium.App are children of the Titanium module. Titanium.App itself has a child, Titanium.App.Properties.
- An alternate ID can be provided for use in require() statements through the @Kroll.module#id annotation element.

Proxies

- Must extend KrollProxy and have the @Kroll.proxy annotation.
- Can automatically generate a "create" method on a parent module by using the @Kroll.proxy#creatableInModule annotation element. For example, Titanium.UI.createView() is the result of this annotation on ViewProxy. UIModule does not define a createView method.
- Have built-in event management primarily through KrollProxy#fireEvent and KrollProxy#hasListeners.
- Wrap native views so JavaScript can interact with them. For example, when you create a native view by calling Titanium.UI.createView() in JavaScript, the returned object is actually a ViewProxy that connects JavaScript to the TiView.

Methods

Methods of a proxy or module are exposed with the @Kroll.method annotation. A simple example:

Properties

Properties are exposed as a pair of getter and setter methods with the @Kroll.getProperty and @Kroll.setProperty annotations.

Constants

A constant is simply a static property on a @Kroll.module. Annotate the property with @Kroll.constant and declare it as both `static` and `final`. Here's an example

Adding Views

Views in Titanium must have two classes:

- The view proxy: A subclass of TiViewProxy.
  - Responsible for exposing methods and properties of the view to JavaScript (just as a normal proxy would do).
  - Implements `TiUIView createView(Activity activity)` which returns a new instance of TiUIView.
- The view implementation: A subclass of TiUIView.
  - Must call `setNativeView` with an instance of View either in the constructor, or in `processProperties`.
  - The view implementation is responsible for taking data from the view proxy, and applying it directly to the native View that it exposes.
  -
  - **Getting the Current Activity**
  - To get access to the current activity, first use TiApplication's getInstance() method, and then use the getCurrentActivity() method:

```
TiApplication appContext = TiApplication.getInstance();
```

```
    Activity activity = appContext.getCurrentActivity();
```

IOS Step 2: Basic Module Architecture

The Module architecture contains the following key interface components:

Proxy	A base class that represents the native binding between your JavaScript code and native code.
ViewProxy	A specialized Proxy that knows how to render Views.
View	The visual representation of a UI component which Titanium can render.
Module	A special type of Proxy that describes a specific API set, or namespace.

When building a Module, you can only have one Module class but you can have zero or more Proxies, Views and ViewProxies.
For each View, you will need a ViewProxy.

The ViewProxy represents the model data (which is kept inside the proxy itself in case the View needs to be released) and is responsible for exposing the APIs and events that the View supports.

You create a Proxy when you want to return non-visual data between JavaScript and native. The Proxy knows how to handle any method and property dispatching and event firing.

Proxy

To create a Proxy, your interface declaration must extend TiProxy. Import "TiProxy.h" in your import statement. For example, your interface would be:

```
#import "TiProxy.h"
@interface FooProxy : TiProxy
{
}
@end
```

Proxy Methods

Proxies expose methods and properties by simply using standard Objective-C syntax. To expose a method, a Proxy must have one of the following valid signatures:

```
-(id)methodName:(id)args
{
}
```

Memory Management
Proxies act like any Objective-C class and all memory management rules must be considered. When returning a new proxy instance from a method, you must autorelease the instance. Titanium will retain a reference to the proxy which maps to a reference to the resulting JavaScript variable reference. However, once the JavaScript variable is no longer referenceable, it will be released and your proxy will be sent the dealloc message.

You must take special care to retain/release your objects in Titanium just like you would in any Objective-C based programs. Improper retain/release will cause crashes and undesired results.

View Proxy

A View Proxy is a specialization of a Proxy that is used for Views — objects which interact with the UI to draw things on the screen.

The View Proxy holds the data (model) and acts like a controller for dispatching property changes and methods against the view. The View handles the internal logic for interacting with UIKit and handling UIKit events. The View is a model delegate to the View Proxy and, as long as referenced, receives property changes.

The View Proxy does not always retain a valid reference to a View. The View is only created on demand, as needed. Once the View is created and retained by the View Proxy, all property change events on the View Proxy will be forwarded to the View.

View

A View implementation must extend the TiUIView class. The TiUIView class extends UIView and provides Titanium specific functionality.