Hyperloop for Android Programming Guide

> *This documentation is made available before final release and is subject to change without notice and comes with no warranty express or implied.*

Requirements

You'll need to have the following minimum requirements to use Hyperloop for Android:

- Titanium 5.2.0
- Android 2.3.3+ SDK

Pre-release Installation

For pre-release, you'll need to update to the latest unreleased version of Titanium 5.2.0 by running `ti sdk install -b master -d`. Make sure you set the version of your application to use this version in your `tiapp.xml` `<sdk-version>`.

Classes

Overview

Classes in Hyperloop map to the underlying classes defined in Java. For example, if you have a class such as `android.view.View` defined, you would reference it using a standard require such as:

```
var View = require('android.view.View');
```

This will return the `View` class object. Meaning, it's not an instance of a `View`, but the `View` class itself.

Once you have a the Class reference returned from `require`, you can call normal JavaScript property and functions against it. Remember, at this point calling functions or properties against the class object above will be accessing Class level (static) Java methods (not instance

level).

For example, you could get the generated view id of the `View` using the example:

```
var generatedId = View.generateViewId();
```

This is because `generateViewId` is defined as a static method.

Instantiation

To instantiate a native Class and create an instance, you can use `new` just as you normally do in Javascript or Java:

```
var view = new View(activity);
```

Methods and Fields

Methods in Java are mapped to JavaScript functions. Fields in Java are mapped to JavaScript property accessors. static methods or fields (such as constants) will be attached to the class type.

For example:

```
public class Example {
    public int field;
    public static final String staticString = "";
    public void method(int argument);
    public static boolean staticMethod();
}
```

Would map to the following in JavaScript:

```
example.field = 123;
Example.staticString;
example.method(567);
var result = Example.staticMethod();
```

Method resolution

If a class has overloads for a method (multiple forms of the method with different signatures, but the same name), we will attempt to match the correct method to invoke on the Java side by matching the passed in arguments to the closest match. Typically, this involves matching the name, number of arguments and the ability to convert the passed in arguments (in-order) to the method's parameter types. We are slightly more liberal in accepting numeric primitives than typical method resolution due to the conversion of JS Numbers.

Casting

Sometimes interfaces define generic return types such as `Object` and you will need to cast them to a different type to then reference methods and properties of the class. You can pass along the object you want to wrap to the constructor of the type you want to wrap it in.

For example, suppose the result of the function returned an `Object` but you know the implementation is actually a `View`. You could use the following:

```
var view = new View(object);
// call View instance methods on view variable
```

Be careful with casting: If you cast an object which is actually something different, you will experience an error and likely a crash.

You can also cast a Titanium UI Component into its equivalent. For example, this would work:

```
var tiView = Ti.UI.createView( { backgroundColor : "red" } );
var nativeView = new View(tiView);
console.log('X (relative to parent): ', nativeView.getLeft());
```

Interfaces

Interfaces may be implemented using a Javascript syntax similar to an anonymous Java class. Call the constructor of the interface type with a JS object that contains properties that match the interface method names, and corresponding values as function that implement them.

For example, to create an instance that implements `android.view.View.OnTouchListener`:

```
var OnTouchListener = require('android.view.View.OnTouchListener'),
    listener = new OnTouchListener({
        onTouch: function(v, event) {
            // Do some work here
            return true;
        }
    });
```

Creating your own classes

Currently, extending Java classes from Javascript is not supported in the latest version.

Using Third-party libraries

You can use Third-party libraries in Hyperloop.

JARs

Simply place the JAR files into the `platform/android` folder of your app. Hyperloop will pick up the JAR files and will generate necessary bindings and include the JARs in your app.

AARs

Simply place the AAR files into the `platform/android` folder of your app. Hyperloop will pick up the AAR files and will generate necessary bindings, extract resources, extract and use the classes.jar, *.so file, etc.