How to use an optional API in Java ME

This article demonstrates how to detect the presence of an optional API (such as the Location API) at run-time, and use it only if it is available. For example, to develop an application that works on device without the Location API, but can become location-aware on devices that *do* have the Location API, without requiring multiple builds.

This example will refer to the Location API, but applies equally to other APIs.

Caveat

This technique should work on any Java virtual machine that complies with the VM specification (including the CLDC specification). However, there are some VMs that perform some kind of install-time verification, optimization or re-compilation of the Java bytecode, which prevent this technique from working. It is known not to work on the Motorola T720 (an old, MIDP-1 device), and on BlackBerry devices (BlackBerry devices load all classes at application start-up, to check for access to secured APIs).

Theory

In order to execute the code in a Java class, five steps must be completed.

1. **Loading.** The .class file is located, and loaded into the Java heap. This happens either when the class is needed by the *resolution* process of another class, or when Class.forName() is invoked.
2. **Verification.** The byte-code is verified to ensure that represents a correct and valid Java program. This always occurs after *loading* and before *preparation*.
3. **Preparation.** Memory is allocated for the static fields of the class, and initialized to default values (0, 0.0, '\0', false or null). This always occurs before *resolution*.
4. **Resolution.** Any other classes referenced by this class are located and loaded (but *not* classes named only in the string argument to Class.forName()).
5. **Initialization.** The class-initializer code is executed. This includes any static{} blocks in the class definition, and any code used to initialize static fields. This step is triggered by the first attempt to create an instance of the class, to access one of its static fields, or to invoke one of its static methods.

Resolution is the key step. If we use a class that references a missing API, this is the point where the code will break. We need to make sure that this process only takes place if the API is present.

I didn't say when resolution takes place. That's because it varies between different JVM implementations. Different VMs perform this step at different times. Here are the two extremes:

- **Earliest**: The VM performs verification, preparation and full resolution of the class immediately after loading. This usually results in a whole series of classes being loaded and resolved, often the entire application.
- **Latest**: The VM might wait until the external class references need to be resolved in order to execute code in the class This may occur *after* initialization, if the referenced class is not needed to execute the initializer code.

And there are variations in between these two.

On VMs that use late resolution, it may be enough not to execute any code that refers to the missing API. However, such code may break on VMs that use early resolution.

To be sure our code will work, we need to make sure that no class is ever **loaded** that refers to the optional API, unless that API is present.

Technique

Here is one solution. For the sake of example, we'll be using the Location API. This technique can be applied to any API in the same way.

Create a Separate Package

This is optional, but makes the technique "safer". Let's say our application is in the package:

```
package com.mycompany.myapplication;
```


(C) Copyright Nokia 2014. All rights reserved.

No classes in this package will have any reference to the location API. We'll put all the location services code in a separate package:

```
package com.mycompany.locationservices;
```

By doing this, we can stop code in the application from accessing location code directly, using package-privacy. This will make more sense later.

Create An Abstract Class

We need to create one abstract class in the location services package. This will be the only public class in that package, and is the only one that *must not* use any part of the location API. It will define an interface through which we will use location services.

```
package com.mycompany.locationservices;

public abstract class LocationProvider {
    // define the interface through which we'll access location information
    public abstract double getLatitude();
    public abstract double getLongitude();

    /**
     *  this method will be used to get access to a LocationProvider class
     */
    public static LocationProvider getProvider() throws ClassNotFoundException {
        LocationProvider provider;
        try {
            // this will throw an exception if JSR-179 is missing
            Class.forName("javax.microedition.location.Location");

            // more about this class later
            Class c =
Class.forName("com.mycompany.locationservices.LocationImplementation");
            provider = (LocationProvider)(c.newInstance());
        } catch (Exception e) {
            throw new ClassNotFoundExcepion("No Location API");
        }
        return provider;
    }
}
```

This class makes no reference to JSR-179 classes, nor to any other class from the locationservices package, *except* for references in Strings in calls to Class.forName(). These will not cause classes to load until they are executed, and then they will throw exceptions (which can be caught and handled) if there is a problem.

Since this will be the only public class in the package, it will be the only class we can refer to from the main application. Because it doesn't reference any location classes, it's safe to reference it.

The LocationImplementation class will not be public, and so must be in the same package as LocationProvider to avoid an IllegalAccessError.

Create An Implementation

Now, we need the aforementioned LocationImplementation class.

```
package com.mycompany.locationservices;
```


(C) Copyright Nokia 2014. All rights reserved.

```
import javax.microedition.location.*;

class LocationImplementation extends LocationProvider {

    LocationImplementation() {
        /*
            We must have a no-parameter constructor, or Class.newInstance() cannot
            create an instance of this class
        */
    }

    // implement the abstract LocationProvider methods

    public double getLatitude() {
        // do whatever is needed
    }

    public double getLongitude() {
        // do whatever is needed
    }
}
```

Note that this class is not public. This prevents classes outside this package from using it, and is a safe guard against accidentally using it elsewhere (and ruining our cunning plan).

Any other classes needed to support this can be created in this package, but make them package-private (not public) too.

We could equally have done this by creating a package-private implementation of a public interface. I chose an abstract class simply because it gives us somewhere to put the static method for getting the implementation class.

Using The LocationProvider

Using this in the application is now easy.

```
try {
    LocationProvider provider = LocationProvider.getProvider();
    double latitude  = provider.getLatitude();
    double longitude = provider.getLongitude();
} catch (ClassNotFoundException cnfe) {
    // no location API on this device
}
```


(C) Copyright Nokia 2014. All rights reserved.

(C) Copyright Nokia 2014. All rights reserved.