Cocoa-Java Integration Guide

(Legacy)

Cocoa > Java

2006-10-03

Contents

Introduction to Cocoa-Java Integration Guide

Important: The Cocoa-Java API is deprecated in Mac OS X version 10.4 and later. You should use the Objective-C API instead; this API is documented in *Application Kit Framework Reference* and *Foundation Framework Reference*.

This document discusses issues that arise when writing Java applications with Cocoa, which is implemented in Objective-C.

Important: Features added to Cocoa in Mac OS X versions later than 10.4 will not be added to the Cocoa-Java programming interface. Therefore, you should develop Cocoa applications using Objective-C to take advantage of existing and upcoming Cocoa features.

Organization of This Document

Here are the concepts covered:

- "Method Selectors" (page 7) discusses how Java can make use of Cocoa's target-action paradigm using the NSSelector class.
- "Java Memory Management" (page 9) discusses how Java sometimes has to take Objective-C's memory management into account when using Cocoa.

Here are the tasks covered:

- "Using NSSelector" (page 11) describes how you use the Java-only class NSSelector to implement target-action behavior.

Method Selectors

Cocoa makes extensive use of the target-action design paradigm. This allows you to have one object send an arbitrary message (the action) to an arbitrary object (the target) in response to an event, such as a button being pressed or a notification being posted. The Objective-C language provides support for this paradigm by allowing any message to be sent to any object, without needing to know the class of the object first. The message is specified by the Objective-C type `SEL` (also called a selector), usually obtained with the `@selector()` directive. (See "Selectors" in *The Objective-C Programming Language* for a more detailed discussion.)

Java, however, is a strongly-typed language wherein you must know the class of an object before sending it a message. To bring target-action support to Java, Cocoa implements the NSSelector class. An NSSelector object specifies a method signature, which is a method's name and parameter list. You can later apply a selector on any object, and it performs the method that matches the selector, if there is one.

NSSelector is similar to `java.lang.reflect.Method`, which fully specifies a particular class's implementation of a method, but you can apply it only to objects of that class. NSSelector does not specify the method's class, so you can apply it to an object of any class. To find the `java.lang.reflect.Method` object for a method that matches an NSSelector object and that is in a particular object or class, use the NSSelector instance method `methodOnObject` or `methodOnClass`.

See "Using NSSelector" (page 11) for examples of how you use the NSSelector class.

Java Memory Management

Under most circumstances, Java applications do not need to concern themselves with memory management. Because Cocoa is an Objective-C framework without automatic garbage collection, however, Java applications sometimes need to take steps to handle peculiarities in this mixed-language environment.

Creating Cocoa Objects

Creating and destroying Cocoa objects in Java is the same as creating and destroying any object in Java. You create an object with a constructor and the object persists as long as a reference is maintained. When the last reference is eliminated, the object is automatically destroyed. The Java Bridge, which handles the communication between the Objective-C and Java environments, works with the Java garbage collector to track references to Objective-C objects in the Java environment (and Java objects in the Objective-C environment) to make sure the objects persist as long as references exist. (See "Weak References in Java" (page 9) for a special case when this is not strictly true.)

Also note that although Cocoa object constructors do not declare `throws` clauses, they can still throw a runtime exception. The thrown exception is an instance of the NSException class, a subclass of `java.lang.RuntimeException`.

Weak References in Java

Objective-C does not implement automatic reference counting, so applications need to explicitly `retain` objects that need to persist and `release` objects when a reference is no longer needed. In some cases, however, Cocoa uses a weak reference to objects wherein a pointer to an object is recorded but the object is not retained. (See "Weak References to Objects" for details.)

Weak references are mostly used to break circular references where retaining each object would prevent the objects from deallocating when they should. Child-parent relationships, such as in the view hierarchy, are cases where weak references are used. Cocoa also uses weak references for table data sources, outline view items, notification observers, and miscellaneous targets and delegates.

Weak references pose a problem for Java applications when a pure Java object, one not descended from Cocoa's NSObject, is passed to a Cocoa object using a weak reference. When the Java object is passed through the Java bridge to the Objective-C Cocoa object, a proxy is created to represent the Java object. Because only a weak reference is used, when control passes back to the Java side, the proxy object is left with a reference count of zero, so it is deallocated. Messages sent to that object, messages that should be forwarded to the Java object, instead find a deallocated object, so an error occurs.

To work around the problem, the proxy object needs to get retained by an Objective-C object. The easiest way to do this is to store the Java object inside an NSArray. An NSArray, an Objective-C object exported into Java, retains its elements, releasing them when the array is deallocated. The Java application thus needs to keep a reference to the NSArray in addition to (or perhaps instead of) the pure Java object.

Using NSSelector

To create a selector in Java, use NSSelector's single constructor, which takes the method's name and an array of the parameter types. Note that to obtain a Class object for a type, append `.class` to the type's name. For example, the Class object for NSObject is `NSObject.class` and the Class object for boolean is `boolean.class`.

This code sample creates a selector for the `doIt` method:

```
void doIt(String str, int i) { . . . }
NSSelector sel =
    new NSSelector("doIt", new Class[] {String.class, int.class} );
```

To apply a selector on an object, use the overloaded instance method `invoke`. It performs the method that matches the selector and returns the result. If the target object does not have a method matching the selector, it throws NoSuchMethodException. The most basic form of `invoke` takes the target object and an Object array of the arguments. Other forms are convenience methods for selectors with no, one, or two arguments. Note that to pass an argument of a primitive type to `invoke`, use an object of the corresponding wrapper class. `invoke` converts the object back to the primitive type when it invokes the method. For example, to pass the float `f`, use `new Float(f)`; and to pass the boolean value `true`, use `new Boolean(true)`.

This code sample gives you two ways to apply the selector `sel` (defined above) to an object:

```
MyClass obj1 = new MyClass(), obj2 = new MyClass();
int i = 5;
sel.invoke(obj1, new Object[] { "hi", new Integer(i) });
sel.invoke(obj2, "bye", new Integer(10));
```

To create and apply a selector in one step, use the overloaded static method `invoke`. The basic form takes four arguments: the method name, an array of the parameter types, the target object, and an array of the arguments. Other forms are convenience methods for selectors with one or two arguments. This code sample shows two ways to create and apply a selector for the `doIt` method:

```
void doIt(String str, int i) { . . . }
MyClass obj1 = new MyClass(), obj2 = new MyClass();
int i = 5;

NSSelector.invoke("doIt", new Class[] {String.class, int.class},
    obj1, new Object[] {"hi", new Integer(i)});
NSSelector.invoke("doIt", String.class, int.class,
    obj1, "bye", new Integer(10));
```

Other methods return whether an object or class has a method that matches a selector (`implementedByObject` and `implementedByClass`) and returns the method name and parameter types for a selector (`name` and `parameterTypes`).

Document Revision History

This table describes the changes to *Cocoa-Java Integration Guide*.

Date	Notes
2006-10-03	Deprecated the Cocoa-Java Integration Guide.
2005-07-07	Updated to specify the purpose of Cocoa-Java technology in Mac OS X. Changed title from "Java/Objective-C Language Integration."
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.