

DigitalPersona[®] **Gold**

SDK for Java

Programmer's Guide
Version 3.2.0



DigitalPersona, Inc.

© 2005 DigitalPersona, Inc. All Rights Reserved.

All intellectual property rights in the DigitalPersona software, firmware, hardware and documentation included with or described in this Guide are owned by DigitalPersona or its suppliers and are protected by United States copyright laws, other applicable copyright laws, and international treaty provisions. DigitalPersona and its suppliers retain all rights not expressly granted.

U.are.U®, DigitalPersona® and One Touch® are registered trademarks of DigitalPersona, Inc.

Windows, Windows 2000, Windows 2003 and Windows XP are registered trademarks of Microsoft Corporation. All other trademarks are property of their respective owners.

This DigitalPersona Gold SDK for Java Programmer's Guide and the software it describes are furnished under license as set forth in the Installation Software screen "License Agreement." Except as permitted by such license, no part of this document may be reproduced, stored, transmitted and translated, in any form and by any means, without the prior written consent of DigitalPersona. The contents of this manual are furnished for informational use only and are subject to change without notice. Any mention of third-party companies and products is for demonstration purposes only and constitutes neither an endorsement nor a recommendation. DigitalPersona assumes no responsibility with regard to the performance or use of these third-party products. DigitalPersona makes every effort to ensure the accuracy of its documentation and assumes no responsibility or liability for any errors or inaccuracies that may appear in it. This document is subject to the DigitalPersona LIMITED WARRANTY and other general provisions set forth in the Appendix of this manual.

Should you have any questions concerning this document, or if you need to contact DigitalPersona for any other reason, write to:

DigitalPersona, Inc., 720 Bay Road, Suite 100, Redwood City, CA 94063 USA

Table of Contents

1 Introduction	1
Requisite Knowledge	1
Other Support Resources	1
Typographic Conventions	2
Notational Conventions	2
Naming Conventions	3
Your Feedback Requested	3
2 API Overview	5
Getting Started with the Java SDK	5
Registration	6
Using the XTF Registration Template	7
Verification	8
Acquiring Templates	10
Template Matching	11
Template Searching	12
Priority Level	12
Fingerprint Scans	13
Deploying Applications Built with the Gold SDK	13
3 Appendix A: Java SDK API	15
Interface DpFpEventHandler	15
Fields	15
Methods	17
Class DpFpImage	17
Constructor	17
Methods	17
Class DpFpTemplate	17
Constructor	18
Methods	18
Class DpFpRegistration	18
Constructor	18
Methods	18
Class DpFpRegistrationException	19
Constructor	19

Class DpFpVerification	19
Constructor	19
Methods	19
Class DpFpVerificationException	20
Constructor	20
Class DpFpTemplateAcquisition	20
Constructor	20
Methods	20
Class DpFpTemplateAcquisitionException	21
Constructor	21
Class DpFpTemplateMatching	21
Constructor	21
Methods	21
Class DpFpMatchException	22
Constructors	22
Class DpFpMatchResult	22
Constructor	22
Methods	22
Fields	22
Class DpFpToolkitException	23
Constructors	23
Class DeviceInfo	23
Constructor	23
Methods	23
Class DpFpFingerprintToolkit	24
Constructor	24
Fields	24
Methods	25
4 Appendix B	29
Fingerprint Reader Usage and Maintenance	29
Proper Fingerprint Reader Usage	29
Cleaning the Reader	30

5 Appendix C	31
Regulatory Information	31

Introduction

1

The DigitalPersona Gold SDK for Java Programmer's Guide provides guidelines for adding fingerprint authentication functionality to Java applications running on WIN32 platforms. In addition, it provides a complete listing of the Java SDK API.

This chapter describes the requisite knowledge a Java programmer must possess in order to use this guide and the SDK. It also explains your technical support options, as well as the conventions used in this guide.

Following is a description of the remaining chapters:

Chapter 2, API Overview, provides instructions for registering fingerprints for authentication and verifying them using the DigitalPersona Gold SDK for Java API.

Appendix A, Java SDK API, lists all classes, interfaces, exceptions and fields of the DigitalPersona Gold SDK for Java API, including brief descriptions for some.

Requisite Knowledge

Any programmer using the DigitalPersona Gold SDK for Java must be proficient in several areas of the Java programming language (version 1.2 or higher); specifically, Java data types, event handling and exception handling.

Support Resources

In addition to this guide, the following resources are provided for additional support to programmers using the DigitalPersona Gold SDK for Java:

- A Readme file is provided on the product CD, which contains last-minute information about the product.
- The DigitalPersona Web site (<http://www.digitalpersona.com>) provides an online technical support form in the Support section. You can describe your issue and include your contact information and a technical support representative will contact you by e-mail or phone.
- E-mail support is available at techsupport@digitalpersona.com.
- Phone support can be reached at (877) 378-2740 in the U.S. only. Outside the U.S., call +1 650-474-4000.

Typographic Conventions

The following typographic conventions are used in this guide:

- `Courier` indicates text that is typed by the user.

Example:

“Type `http://www.digitalpersona.com/` in the Address text box.”

You would only type “`http://www.digitalpersona.com/`” and would not type any surrounding text.

- Text in **Courier bold** and surrounded by brackets [] indicates information that is always supplied by you and will vary depending on a particular circumstance.

Example:

“Type `http:// [your company web site URL] /` in the Address text box.”

You would type “`http://`”, then type your company web site URL—not the words “[your company web site URL]”—and then “/”.

It is also used to display information that is dynamically generated by DigitalPersona Pro.

Notational Conventions

The following notational conventions are used in this guide to call attention to information of special importance:

Note

A note highlights information that may help you better understand the text and its concepts.

Warning

A warning advises you that failure to take or avoid a specific action could result in your inability to complete the required tasks or will cause undesired results in the use of the software or hardware.

Naming Conventions

For brevity and easier reading of this guide, the following naming conventions are used to describe the DigitalPersona Gold SDK for Java and fingerprint reader hardware:

- Gold SDK, Java SDK, and SDK sometimes replace the full name, DigitalPersona Gold SDK for Java.
- Reader—in both upper and lower case—is always used without the preceding U.are.U. It replaces the full product name, U.are.U Fingerprint Reader.

Your Feedback Requested

The information in this guide has been thoroughly reviewed and tested. If you find errors or have suggestions for future publications, contact DigitalPersona at:

720 Bay Road, Suite 100
Redwood City, California 94063 USA
(650) 474-4000
(650) 298-8313 FAX

Chapter 1 Introduction

API Overview

2

This chapter provides instructions for adding fingerprint authentication functionality to Java applications to prove someone's identity by providing a mechanism that acquires fingerprint scans, converts them to templates and compares these templates to determine whether they are from the same finger.

Fingerprint templates are highly compressed and digitally encoded mathematical representations of fingerprint scans and are created whenever a user touches the fingerprint reader. They are encoded with a one-way algorithm that cannot be reversed to recreate the image of that fingerprint from its fingerprint template. The fingerprint scans themselves are never stored; rather, they are discarded after the fingerprint template is created. As a result, any Java application can accurately attest to one's identity.

Both processes of fingerprint authentication—registration and verification—are described in the remaining sections of this chapter, as well as overview information and sample code for implementing these processes.

Getting Started with the Java SDK

To develop Java applications using the DigitalPersona Gold SDK for Java package, the `jsdk.jar` and `jsdkdocs.jar` files must be in the Java class path. The Gold SDK installer software places these files in the following directory:

```
[hard drive]:\Program Files\DigitalPersona\Gold SDK
```

The installer also places a file named, `DpjFp.dll`, in the `Windows/System32` directory. Although it does not need to be placed in the Java class path, it must be present for the Java application using the SDK to work properly.

To add fingerprint authentication functionality to a Java application, import the `com.digitalpersona.uareu.toolkit` package:

```
import com.digitalpersona.uareu.toolkit.package.*;
```

When this package is imported, you can add fingerprint registration and verification to an application, as described in the next two sections.

Registration

Fingerprint registration is the process of acquiring four fingerprint scans and deriving from them a template that can be compared with a fingerprint template acquired during verification to determine if they came from the same finger.

This template—called a registration template—is created from four pre-registration templates, derived from fingerprint scans of acceptable quality acquired during the registration process.

At a minimum, a Java application implementing the registration process should follow these behavior guidelines:

- When the registration process is initiated, the user should be prompted to touch the fingerprint reader.
- The user should be given feedback on the quality of the acquired fingerprint scan, indicating whether the scan is acceptable or not.
- The user should be prompted to place a finger on the reader until four acceptable scans are acquired.
- When the registration template is created, the user should be notified that the fingerprint is registered.

To add the fingerprint registration functionality to a Java application

1 In the Java application, create an instance of the Registration class:

```
Registration reg = new Registration();
```

2 Listen for events by passing the class performing the registration process to an instance of the DpFpRegistration class using the setHandler method:

```
reg.setHandler(this);
```

this is any class that implements the DpFpEventHandler interface.

3 Begin the registration process by calling the register method of the DpFpRegistration class:

```
reg.register();
```

There are several events fired during the registration process:

- **EVT_WAITING_FOR_IMAGE**, which you can use to prompt the user to touch the reader.

- **EVT_FINGER_TOUCHING**, when the user places a finger on the reader. In the event handler for this event, you can, for example, play a sound and also display a message instructing the user to remove the finger.
 - **EVT_FINGER_REMOVED**, after a user removes the finger from the reader. If this event is not fired soon after the finger touching event, you can instruct the user to take the finger off the reader.
 - **EVT_IMAGE_RECEIVED**, after the fingerprint scan is acquired, allowing you to display the image, as described in “Fingerprint Scans” on page 13.
 - Various events related to the quality of the scan are fired and are listed in “Interface DpFpEventHandler” on page 15. Use these events to inform the user of the quality of the acquired fingerprint scan.
 - **EVT_TEMPLATE_EXTRACTED** is fired when a good fingerprint scan was successfully converted to a pre-registration template.
- 4 When four good scans are acquired and successfully converted to pre-registration templates, a registration template is created.
- The **EVT_REGISTRATION_COMPLETE** event is fired and the registration template object is passed to the event handler as a **DpFpTemplate** object. You can then store the template where desired, e.g., in a **DpFpTemplate[]** array, database, etc.

The application can cancel the registration by calling the **cancel** method of the registration object. When canceled, the **EVT_REGISTRATION_CANCELED** event is fired. If a system error occurs during the registration process, the **EVT_REGISTRATION_ERROR** event is fired.

Using the XTF Registration Template

The XTF template is an extended form of the registration template and provides improved verification performance. The XTF template contains data from each of the four registration fingerprint scans. By default, the basic registration template is used. Use the **XTFTemplate** methods in the **DpFpRegistration** class to implement the XTF Template instead of the basic template.

Verification time when using the XTF template is on average 10% longer than the basic registration. At the most, verification time with the XTF template can be up to four times longer than the basic template.

The XTF template uses approximately four times more space than the basic template when saved to a file or to a database. Developers who have already created programs using the basic template and want to switch to the XTF template must consider if the size increase will require more space allocated in their existing databases.

Verification

In the verification process, a user touches the reader and a verification template is created from the fingerprint scan. To prove user identity, the verification template is compared with one or more registration templates (see “Registration” on page 6) to determine if they are from the same finger.

A Java application implementing the verification process should follow these behavior guidelines:

- When the verification process is initiated, the user should be prompted to touch the reader with a registered fingerprint.
- If the fingerprint scan is acceptable, the Java application should load stored registration templates and perform a verification.
- The user should then be informed of the match result. If a match is found, the application can grant access to protected functions, data, etc.

To add verification functionality to a Java application

- 1 Load the data blob representing a fingerprint image and create a template:

```
DpFpTemplate regTemplate = new DpFpTemplate();  
regTemplate.setData(blob);
```

- 2 Add the template to an array of DpFpTemplate[] objects by passing the registration template data blob to an instance of DpFpTemplate using its setData() method:

```
DpFpTemplate[] regTemplates = new DpFpTemplate[];  
regTemplates[x] = regTemplate;
```

x is the array index number.

- 3 Create an instance of the Verification class:

```
DpFpVerification ver = new DpFpVerification();
```

- 4 Listen for DpFpVerification events by passing a reference to the calling class to the DpFpVerification instance using the setHandler method:

```
ver.setHandler(this);
```

- 5 Initiate the verification process by calling the verify method on the instance of the DpFpVerification object and passing the DpFpTemplate array to it:

```
ver.verify(regTemplates);
```

There are several events fired during the verification process:

- **EVT_WAITING_FOR_IMAGE**, which you can use to prompt the user to touch the reader.
- **EVT_FINGER_TOUCHING**, when the user places a finger on the reader. In the event handler for this event, you can, for example, play a sound and display a message instructing the user to remove the finger.
- **EVT_FINGER_REMOVED**, after a user removes the finger from the reader. If this event is not fired soon after the finger touching event, you can instruct the user to take the finger off the reader.
- **EVT_IMAGE_RECEIVED**, after the fingerprint scan is acquired, allowing you to display the image, as described in “Fingerprint Scans” on page 13.
- Various events related to the quality of the scan are fired and are listed in “Interface DpFpEventHandler” on page 15. Use these events to inform the user of the quality of the acquired fingerprint scan.
- **EVT_TEMPLATE_EXTRACTED** is fired when a good fingerprint scan was successfully converted to a verification template.

- 6 When the image is converted into a verification template, the **EVT_VERIFICATION_COMPLETE** is fired and the template object is passed to it. In the event handler, you can perform the match by creating an instance of the MatchResult class, using the template object:

```
MatchResult matchRes = templateObj;
```

templateObj is an instance of the DpFpTemplate class.

- 7 Then, using the `getResult()` method, you can determine whether there was a match between a template in the array of registration templates and the verification template. For example:

```
if (matchRes.getResult() == MatchResult.FT_SUCCESS)...
```

To cancel the verification operation, call the `cancel` method of the verification object. When canceled, the `EVT_VERIFICATION_CANCELED` event is fired. If a system error occurs during the verification operation, the `EVT_VERIFICATION_ERROR` event is fired.

Acquiring Templates

In some cases, programmers will want to acquire templates without performing the verification and registration operations.

To acquire templates without performing registration or verification

- 1 Create an instance of the `DpFpTemplateAcquisition` class:

```
DpFpTemplateAcquisition template =  
new DpFpTemplateAcquisition();
```

- 2 Listen for template acquisition events by passing the class performing the acquisition to an instance of the `DpFpTemplateAcquisition` class using the `setHandler` method:

```
template.setHandler(this);
```

`this` is any class that implements the `DpFpEventHandler` interface and contains an instance of the `DpFpTemplateAcquisition` class.

- 3 Begin the acquisition process by calling the `acquireTemplate` method of the `DpFpTemplateAcquisition` class:

```
template.acquireTemplate();
```

There are several events fired during the verification process:

- **EVT_WAITING_FOR_IMAGE**, which you can use to prompt the user to touch the reader.
- **EVT_FINGER_TOUCHING**, when the user places a finger on the reader. In the event handler for this event, you can, for example, play a sound or display a message instructing the user to remove the finger.

- **EVT_FINGER_REMOVED**, after a user removes the finger from the reader. If this event is not fired soon after the finger touching event, you can instruct the user to take the finger off the reader.
 - **EVT_IMAGE_RECEIVED**, after the fingerprint scan is acquired, allowing you to display the image, as described in “Fingerprint Scans” on page 13.
 - Various events related to the quality of the scan are fired and are listed in “Interface DpFpEventHandler” on page 15. Use these events to inform the user of the quality of the acquired fingerprint scan.
 - **EVT_TEMPLATE_EXTRACTED** is fired when a good fingerprint scan was successfully converted to a verification template.
- 4 When the user submits a good fingerprint scan, it is converted to a `DpFpTemplate` object and passed to the event handler of the `EVT_TEMPLATE_ACQUISITION_COMPLETE` event. You can then store the template or pass it to other applications—such as an authentication program that resides on another server—for storing, matching, etc.

To cancel the template acquisition operation, call the `cancel` method of the `DpFpTemplateAcquisition` object. When canceled, the `EVT_TEMPLATE_ACQUISITION_CANCELED` event is fired. If a system error occurs during the template acquisition operation, the `EVT_TEMPLATE_ACQUISITION_ERROR` event is fired.

Template Matching

With the DigitalPersona Gold SDK for Java, programmers can match templates outside of the fingerprint verification operation (as described in “Verification” on page 8).

To match a verification template to a registration template(s)

- 1 Load each data blob representing a registration template and create a `DpFpTemplate` object:

```
DpFpTemplate regTemplate = new DpFpTemplate();
regTemplate.setData(blob);
```

You must perform this operation for every registration template you want to match and then separately for the verification template:

```
DpFpTemplate verTemplate = new DpFpTemplate();  
verTemplate = blob;
```

- 2 Add each registration template to an array of DpFpTemplate[] objects by passing each one to an instance of DpFpTemplate using its setData() method:

```
DpFpTemplate[] regTemplates = new DpFpTemplate[];  
regTemplates[x] = regTemplate;
```

x is the array index number.

- 3 Create an instance of the DpFpTemplateMatching class:

```
DpFpTemplateMatching tm = new DpFpTemplateMatching();
```

- 4 Initiate the matching process by calling the templateMatch method on the instance of the DpFpTemplateMatching object, passing the DpFpTemplate array and the verification template to it:

```
MatchResult matchRes = tm.templateMatch(verTemplate,  
regTemplates);
```

When called, this method returns a DpFpMatchResult object.

- 5 Use the getResult() method of the newly created DpFpMatchResult object to determine whether there was a match between a registration templates in the DpFpTemplate[] array and the verification template. For example:

```
if (matchRes.getResult() == MatchResult.FT_SUCCESS)...
```

Template Searching

You can implement a more complete search process using the SearchLevel methods. When searching for a matching template, the search process can be complete after a single match occurs, or it can be continued so that each template that matches is found and ranked for quality. The matching results are sorted by order of best match. Use the methods in DpFpVerification and DpFpTemplateMatching to set template searching.

Priority Level

Applications can be assigned a priority level to make them more available to accept the fingerprint information over other applications. High priority means that fingerprint events are sent to the application even though it might not be the active application at the time of the fingerprint input. Normal priority means that

fingerprint events are sent to the application if its window is currently active. Low priority means that fingerprint events are sent to the application if no other application uses the information.

You can have only one application per machine, or per terminal session, assigned to a high priority level and a low level priority. All other applications are considered normal priority. Assigning high or low priority to an application takes the priority from the last application that was assigned that priority, which means it will stop receiving any fingerprint events.

The high and low priorities are already used by the DigitalPersona Pro applications. In some cases, a developer might need to reassign these priorities, but this can stop both DigitalPersona Pro and the custom application from working. It is recommended to run only one application per machine that uses high or low priorities. Use the methods in `DpFpRegistration`, `DpFpVerification`, and `DpFpTemplateAcquisition` to set priority level.

Fingerprint Scans

By importing `java.awt.Image`, you can access and manipulate the fingerprint scan acquired during the registration and verification processes. When a scan is acquired by the reader, the `EVT_IMAGE_RECEIVED` event is fired and an `com.digitalpersona.uareu.toolkit.Image` object is passed to the handler for this event. To convert the image object to a `java.awt.Image` object, call the `getJavaImage()` method:

```
Image image = DpFpImage.getJavaImage();
```

`DpFpImage` is the `com.digitalpersona.uareu.toolkit.Image` object passed to the event handler for the `EVT_IMAGE_RECEIVED` event. You can then display the image using the methods of the `java.awt.Image` class.

Deploying Applications Built with the Gold SDK

Applications built with the DigitalPersona Gold SDK for Java are deployed the same way as all other Java applications, providing that the `jsdk.jar` and `jsdkdocs.jar` are included in the build. Your installer must do one of the following:

- Place the DpjFp.dll file in the Windows/System32 directory on the user's PC.
- Install the DigitalPersona Gold, Fingerprint Recognition Software.



Warning

Failure to install the DpjFp.dll file on the user's PC will cause a fingerprint authentication-enabled application to fail.

Appendix A: Java SDK API

This appendix contains a quick reference to the `com.digitalpersona.ureu.toolkit` package. This package contains all the components necessary to perform the registration and verification processes.

Interface `DpFpEventHandler`

This interface provides event handling feedback for events related to the registration, verification and template acquisition operations, as well as reader-related events.

Fields

<code>static int</code>	<code>EVT_DEVICE_CONNECTION_ERROR</code> General device connection error occurred.
<code>static int</code>	<code>EVT_FINGER_REMOVED</code> User removed finger from reader.
<code>static int</code>	<code>EVT_FINGER_TOUCHING</code> User touched reader with finger.
<code>static int</code>	<code>EVT_IMAGE_RECEIVED</code> Fingerprint scan successfully acquired by reader. Returns a <code>DpFpImage</code> object.
<code>static int</code>	<code>EVT_AREA_TOO_SMALL</code> Acquired fingerprint scan is too small to extract features; the full fingerprint was not placed on the reader.
<code>static int</code>	<code>EVT_NO_CENTRAL_REGION</code> The fingerprint scan does not contain an adequate portion of the center of the fingerprint.
<code>static int</code>	<code>EVT_NOISY_IMAGE</code> Fingerprint scan was too grainy; possibly, too much dirt on the reader window.
<code>static int</code>	<code>EVT_FINGER_OFF_CENTER_HIGH</code> Fingerprint scan is too high.
<code>static int</code>	<code>EVT_FINGER_OFF_CENTER_LEFT</code> Fingerprint scan is too far left.

static int	EVT_FINGER_OFF_CENTER_LOW Fingerprint scan is too low.
static int	EVT_FINGER_OFF_CENTER_RIGHT Fingerprint scan is too far right.
static int	EVT_NOT_ENOUGH_FEATURES Fingerprint scan does not contain enough features to extract a template; the reader window may be cloudy.
static int	EVT_REGISTRATION_CANCELED Registration operation canceled by application.
static int	EVT_REGISTRATION_COMPLETE Registration process complete. Returns a DpFpTemplate object (registration template).
static int	EVT_REGISTRATION_ERROR A system error during registration process.
static int	EVT_IDENTIFICATION_CANCELED Identification operation canceled by application.
static int	EVT_IDENTIFICATION_COMPLETE Identification process complete.
static int	EVT_IDENTIFICATION_ERROR A system error during identification process.
static int	EVT_VERIFICATION_CANCELED Verification process canceled by application.
static int	EVT_VERIFICATION_COMPLETE Verification process complete. Returns a DpFpMatchResult object.
static int	EVT_VERIFICATION_ERROR A system error during verification process.
static int	EVT_TEMPLATE_ACQUISITION_CANCELED Template acquisition process canceled by application.
static int	EVT_TEMPLATE_ACQUISITION_COMPLETE Template acquisition process complete. Returns a DpTpTemplate object (verification template).

```

static int      EVT_TEMPLATE_ACQUISITION_ERROR
                System error during the template acquisition process.
static int      EVT_WAITING_FOR_IMAGE
                The reader is waiting for a fingerprint scan.
static int      EVT_TEMPLATE_EXTRACTED
                A template was successfully extracted from a fingerprint
                scan.
static int      EVT_DEVICE_PLUGGED
                The reader is plugged in.
static int      EVT_DEVICE_UNPLUGGED
                The reader is unplugged.

```

Methods

```
void            dpFpOnEvent(int event, Object obj)
```

Class DpFpImage

```
public class DpFpImage extends java.lang.Object
```

This class represents a fingerprint scan (see “Fingerprint Scans” on page 13).

Constructor

```
DpFpImage(int height, int width)
```

Methods

```
byte[]         getData()
java.awt.Image getJavaImage()
void           setData(byte[] templateData)
```

Class DpFpTemplate

```
public class DpFpTemplate extends java.lang.Object
```

This class represents a fingerprint template as a byte array (see “Registration” on page 6, “Verification” on page 8, “Acquiring Templates” on page 10 or “Template Matching” on page 11).

Constructor

`DpFpTemplate()`

Methods

<code>byte []</code>	<code>getData()</code>
<code>int</code>	<code>getTemplateID()</code>
<code>void</code>	<code>setData(byte [] templateData)</code>
<code>void</code>	<code>setTemplateID(int id)</code>
<code>int</code>	<code>getTemplateType()</code>
<code>void</code>	<code>setTemplateType(int i)</code>

Class DpFpRegistration

```
public class DpFpRegistration
extends java.lang.Object
implements java.lang.Runnable
```

This class provides a mechanism for performing the registration process (see “Registration” on page 6). You can also choose to implement the XTF template (see “Using the XTF Registration Template” on page 7.)

Constructor

`DpFpRegistration()`

Methods

<code>void</code>	<code>cancel()</code>
<code>void</code>	<code>destroy()</code>
<code>void</code>	<code>register()</code>
<code>void</code>	<code>run()</code>
<code>void</code>	<code>setHandler(com.digitalpersona.uareu.DpFpEventHandler handler)</code>
<code>bool</code>	<code>isXTFTemplate()</code>
<code>void</code>	<code>setXTFTemplate(boolean b)</code>
<code>int</code>	<code>getConnectionPriority()</code>

```
void          setConnectionPriority(int priority)
```

Class DpFpRegistrationException

```
public class DpFpTemplate extends java.lang.Exception
```

Constructor

```
DpFpRegistrationException(java.lang.String message)
```

Class DpFpVerification

```
public class DpFpVerification
```

```
extends java.lang.Object
```

```
implements java.lang.Runnable
```

This class provides a mechanism for performing the verification process (see “Verification” on page 8).

Constructor

```
DpFpVerification()
```

Methods

```
void          cancel()
```

```
void          destroy()
```

```
double        getSecurityLevel()
```

```
void          run()
```

```
void          setHandler(com.digitalpersona.uareu.DpFpEventHandler handler)
```

```
void          setSecurityLevel(double d)
```

```
void          verify(com.digitalpersona.uareu.toolkit.DpFpTemplate[] regTemplate)
```

```
void          setConnectionPriority(int priority)
```

```
int           getConnectionPriority()
```

```
bool          getLearning()
```

```
int           getSearchLevel()
```

```
void          setLearning(boolean b)
void          Identify(com.digitalpersona.uareu.toolkit.
                  DpFpTemplate[] regTemplate,
                  java.lang.String threadName)
void          setSearchLevel(boolean b)
```

Class DpFpVerificationException

```
public class DpFpVerificationException
extends java.lang.Exception
```

Exception generated during the verification process.

Constructor

```
DpFpVerificationException(java.lang.String message)
```

Class DpFpTemplateAcquisition

```
public class DpFpTemplateAcquisition
extends java.lang.Object
implements java.lang.Runnable
```

This class facilitates the template acquisition process (see “Acquiring Templates” on page 10).

Constructor

```
DpFpTemplateAcquisition()
```

Methods

```
void          cancel()
void          destroy()
void          run()
void          setHandler(com.digitalpersona.uareu.DpFpEventHandler handler)
void          acquireTemplate()
void          setConnectionPriority(int priority)
```

```
int                getConnectionPriority()
```

Class DpFpTemplateAcquisitionException

```
public class DpFpTemplateAcquisitionException
extends java.lang.Exception
```

Constructor

```
DpFpTemplateAcquisitionException(java.lang.String message)
```

Class DpFpTemplateMatching

```
public class DpFpTemplateMatching
extends java.lang.Object
```

This class performs a match between a set of registration templates and a verification template (see “Template Matching” on page 11).

Constructor

```
DpFpTemplateMatching()
```

Methods

```
double            getSecurityLevel()
void              setSecurityLevel()
DpFpMatchResult  templateMatch(DpFpTemplate verTemplate,
                                DpFpTemplate[] regTemplates)
int              getSearchLevel()
void             setSearchLevel(int n)
bool             getLearning()
void            setLearning(boolean b)
DpFpMatchResult[] templateIdentify(DpFpTemplate
                                    verTemplate, DpFpTemplate[] regTemplates)
```

Class **DpFpMatchException**

```
public class DpFpMatchException
extends java.lang.Exception
```

Constructors

```
DpFpMatchException()
DpFpMatchException(java.lang.String arg0)
DpFpMatchException(java.lang.String arg0,
java.lang.Throwable arg1)
DpFpMatchException(java.lang.Throwable arg0)
```

Class **DpFpMatchResult**

```
public class DpFpMatchResult
extends java.lang.Object
```

This class contains the result of a match between a verification template and a set of registration templates (see “Verification” on page 8 or “Template Matching” on page 11).

Constructor

```
DpFpMatchResult()
```

Methods

```
int          getResult()
int          getScore()
int          getTemplateID()
void         setTemplateID(int i)
double       getFalseAcceptProbability()
```

Fields

```
static int   FT_FAIL
static int   FT_SUCCESS
```

Class DpFpToolkitException

```
public class DpFpToolkitException
extends java.lang.Exception
```

Constructors

```
DpFpToolkitException()
DpFpToolkitException(java.lang.String message)
DpFpToolkitException(java.lang.String arg0,
java.lang.Throwable arg1)
DpFpToolkitException(java.lang.Throwable arg0)
```

Class DeviceInfo

```
public class DeviceInfo extends java.lang.Object
This class represents the device information.
```

Constructor

```
DeviceInfo()
```

Methods

```
int getImgHeight()
com.digitalper getImgResolution()
sona.uareu.toolkit.DeviceInfo.Ft_Image_Resolution
int getImgType()
int getImgWidth()
int getMaxImgSize()
int getNumIntensityLevels()
void setImgResolution(com.digitalpersona.uareu.toolkit.DeviceInfo.Ft_Image_Resolution resolution)
```

```
void          setImgSize(int height, int width)
void          setImgType(int i)
void          setMaxImgSize(int i)
void          setNumIntensityLevels(int i)
```

Class DpFpFingerprintToolkit

```
public class DpFpFingerprintToolkit extends
java.lang.Object
```

This class represents the device information. FingerprintToolkit class provides functions for Initialization, Termination and Settings Management, Reader Handling, Image and Template Acquisition, Registration and Verification.

Constructor

```
DpFpFingerprintToolkit()
```

Fields

```
static int    FT_ALLOW_LEARNING
static int    FT_AREA_TOO_SMALL
static int    FT_BUF_FILLED
static int    FT_DEVICE_CONNECTION_ERROR
static int    FT_DEVICE_ERROR
static int    FT_DEVICE_PLUGGED
static int    FT_DEVICE_UNPLUGGED
static int    FT_ENABLE_MULT_TEMPLATE_REG
static int    FT_FINGER_REMOVED
static int    FT_FINGER_TOUCHING
static int    FT_GOOD_FTR
static int    FT_GOOD_IMG
static int    FT_IMAGE_INFO
static int    FT_IMAGE_READY
static int    FT_IMG_TOO_DARK
static int    FT_IMG_TOO_LIGHT
```

```

static int    FT_IMG_TOO_NOISY
static int    FT_LANDSCAPE
static int    FT_LEAST_PRIORITY
static int    FT_LOW_CONTRAST
static int    FT_NO_CENTRAL_REGION
static int    FT_NOT_ENOUGH_FTR
static int    FT_PORTRAIT
static int    FT_PRE_REG_FTR
static int    FT_READY_TO_FILL_BUF
static int    FT_REG_FTR
static int    FT_REGULAR_PRIORITY
static int    FT_TEMPLATE_INFO
static int    FT_TOP_PRIORITY
static int    FT_UNKNOWN_ERROR
static int    FT_UNKNOWN_FTR_QUALITY
static int    FT_UNKNOWN_IMG_QUALITY
static int    FT_VER_FTR
static int    FT_WAITING_FOR_IMAGE

```

Methods

```

static      FT_acquireImage(long nContext, int
DpFpImage  ftrType, int ImageLen)
static      FT_acquireTemplate(long nContext, int
DpFpTemplate ftrType, int templateLen)

```

This function acquires a scan from the specified reader, checks its quality, and then extracts the template.

```

static void FT_closeContext(long nContext)

```

This function destroys the resources allocated for the given context.

```

static void FT_connectDevice(long nContext, long
devId, int nPriority)

```

This function connects the device.

```

static long      FT_createContext()
                  Create and return id that will identify current usage
                  session.

static void      FT_disconnectDevice(long nContext)
                  This functions releases all the resources allocated to
                  communicate with the reader.

static          FT_getDeviceList(int numDevices)
DeviceInfo[]    This function returns the device info list.

static int       FT_getNumDevices()
                  This function returns the number of devices connected to
                  the machine.

static double    FT_getSecurityLevel(long nContext)
static Len       FT_getTemplateLen(int ftrType, int
regOptions)
                  Returns the minimum and recommended length of
                  template of the given type.

static          FT_identify(long nContext, boolean
DpFpMatchResult doLearning, byte[] regTemplate)
[]

static void      FT_init()
                  This function initializes the toolkit and the reader driver.

static          FT_register(long nContext, int
DpFpTemplate     mcRegOptions, int templateLen)
                  Performs the registration operation.

static void      FT_setSecurityLevel(long nContext, double
                  level)

static void      FT_terminate()
                  This function terminates the use of the dpFpFns module; it
                  releases all the resources.

static          FT_verify(long nContext, boolean
DpFpMatchResult doLearning, byte[] regTemplate)

```

```
static          FT_verifyEx(long nContext, boolean  
DpFpMatchResult doLearning, byte[] regTemplate)  
static void     setupCallback(long nContext, int width,  
int height, int orientation, DpFpImage  
image, java.lang.Object handler)
```

This function sets up the link with the UI to provide feedback and gets the user actions.

