

i-appli Content Developer's Guide for DoJa-3.0

Functional Descriptions

Release 3.0

December 18, 2003

NTT DoCoMo



This product or document is protected by copyright law and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of NTT DoCoMo (including other authors, if any). Third party software, including font technology, is copyrighted and licensed from the suppliers of the software.

Sun, Sun Microsystems, JAVA, J2ME, and J2SE are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. The Sun Logo is a registered trademark of Sun Microsystems, Inc.

i-mode, i-appli and the corresponding logo, Doja and i-melody are trademarks or registered trademarks of NTT DoCoMo.

Other names of companies, products, and services in this document are the trademarks or registered trademarks of their respective companies.

Note that copyrights, trademarks, or registered trademarks are not indicated within this document.

DOCUMENTATION IS PROVIDED “AS IS” AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

Copyright 2000-2003 NTTDoCoMo, Inc. 2-11-1 Nagata-cho, Chiyoda-ku, Tokyo,100-6150, Japan and Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California, 94303, U.S.A. All rights reserved.

This document targets the Java Application Runtime Environment (referred to in this document as the i-appli Runtime Environment) installed in the Mover 505i series. In this manual, the Digital Mover 503i series i-appli Runtime Environment is referred to as the DoJa-1.0 Profile, the Mover 504i series i-appli Runtime Environment as the DoJa-2.0 Profile, and the Mover 505i series i-appli Runtime Environment as the DoJa-3.0 Profile. In cases where the DoJa-2.1 Profile is mentioned, this refers specifically to the i-appli Runtime Environment implemented in the FOMA mobile phones from winter 2002: in cases where the DoJa-2.2 Profile is mentioned, this refers specifically to the i-appli Runtime Environment implemented in the FOMA mobile phones from spring 2003. In addition, in cases where the DoJa-3.5 Profile is mentioned, this refers specially to the i-appli Runtime Environment implemented in the FOMA mobile phones from winter 2003. The profile version of the i-appli Runtime Environment implemented in the mobile phones will be announced by NTT DoCoMo separately.

Please be aware that, even in the case of functions existing from previous profiles, the content of this document does not apply to cellular phones which have support only for earlier profiles. The profiles explained in this manual contain clarification of undefined specifications for pre-existing functions.

Contents

Purpose of This Document	7
Target Audience.....	7
How to Use this Guide	7
Typographical Conventions.....	8
Related Documents	8
Chapter 1 Introduction.....	9
1.1 Benefits of the i-appli Runtime Environment.....	10
1.1.1 Optimized Support for Standalone and Client-server Applications	12
1.1.2 Support for Local Storage of Persistent Data	13
1.2 Java Application Manager	13
1.3 The i-appli Mobile Phone Java Security Model.....	14
1.4 i-appli Installation and Lifecycle	15
1.4.1 Downloading the i-appli	15
1.4.2 Launching the i-appli.....	16
1.4.3 Updating an i-appli	17
1.4.4 Deleting the i-appli	17
Chapter 2 i-appli Runtime Environment.....	18
2.1 i-appli Runtime Environment	18
2.1.1 Native Applications and Operating System.....	19
2.1.2 Java Application Manager.....	19
2.1.3 K Virtual Machine.....	19
2.1.4 CLDC API.....	19
2.1.5 i-appli API	19
2.2 Overview of API in the i-appli Runtime Environment.....	20
2.2.1 Network Architecture	20
2.2.2 CLDC API.....	21
2.2.3 i-appli API	26
Chapter 3 Design Considerations	39
3.1 Characteristics of i-appli Mobile Phone	39
3.2 Memory Issues.....	40
3.2.1 Minimizing Memory Usage	41
3.2.2 Class File Size	42
3.3 Usability and GUI Design	42
3.3.1 Designing for the Target Audience	43
3.3.2 Design Policy for i-appli Operation Methods.....	43
3.3.3 Scrolling	43
3.3.4 Keys and Softkeys	43
3.3.5 Data Entry.....	44
3.3.6 Password Entry.....	44
3.3.7 Use of Threads.....	44
3.4 Security.....	45
3.5 Operating Over a Wireless Network.....	45
3.6 Process Suspension and Resumption.....	45
3.7 Accessing Hardware	47
3.8 Error Processing	47
3.9 Stand-By Applications.....	48
3.10 Identification of Cellular-Phone Models	48

Chapter 4	Designing the User Interface.....	49
4.1	Programming with High-level UI Components.....	52
4.1.1	Using the Panel.....	52
4.1.2	Using the Dialog.....	53
4.1.3	Using Components	53
4.1.4	Placing Components.....	62
4.1.5	Focus and Scrolling.....	65
4.1.6	High Level Event Processing	65
4.1.7	Font Support in Components.....	65
4.2	Programming with Low-level UI API	66
4.2.1	Using the Canvas.....	66
4.2.2	Drawing Graphics.....	67
4.2.3	Rendering to Image Objects	68
4.2.4	Color Support	69
4.2.5	Low-level Event Processing.....	70
4.2.6	Using the IME in the Canvas.....	72
4.3	Event Listeners	73
4.3.1	ComponentListener	73
4.3.2	SoftKeyListener.....	75
4.3.3	KeyListener	75
4.3.4	MediaListener.....	76
4.3.5	TimerListener	76
4.4	Using Multimedia Data	77
4.4.1	Presenter Benefits.....	77
4.4.2	Multiple Simultaneous Playback with AudioPresenter	82
4.4.3	Audio-Synchronization Events.....	83
4.5	Image Encoder.....	83
4.5.1	Image Encoder.....	83
4.5.2	Pixel Operations	85
4.5.3	Image Rotation, Flipping, Enlargement and Shrinking	85
Chapter 5	Controlling Communications.....	86
5.1	Client-Server Programming.....	87
5.1.1	Terminating the Connection.....	90
5.2	Session Management.....	91
5.3	Secure Communication Using HTTPS.....	93
Chapter 6	Text Processing	94
6.1	Text Processing.....	94
6.2	i-Mode Pictographic Characters for i-appli Service.....	95
Chapter 7	Using the ScratchPad and Resources	99
7.1	Reading from and Writing to the ScratchPad	99
7.2	Reading Resources	103
7.3	Error Handling.....	104
Chapter 8	Accessing Platform Resource.....	105
Chapter 9	Stand-By Applications	107
9.1	Overview of Stand-By Applications.....	107
9.1.1	Characteristics of Stand-By Applications.....	107
9.1.2	User Settings for Stand-By Applications.....	109
9.2	Creating Stand-By Applications	110
9.2.1	Stand-By Application Class	110
9.2.2	Life Cycle and Mode Transitions	110
9.2.3	System Events	115
9.2.4	ADF Declarations for Stand-By Applications.....	116
9.3	Launching of Other Functions During Execution of Stand-By Applications.....	116

9.4	Precautions for i-appli API Usage	118
9.4.1	User Interface	118
9.4.2	Input and Output.....	119
9.4.3	Hardware Control.....	119
9.4.4	Application Linking.....	119
Chapter 10	OBEX External Connection.....	120
10.1	OBEX Data Communication.....	121
10.2	OBEX External-Connection API.....	122
10.2.1	OBEX Client API.....	122
10.2.2	OBEX Server API	125
10.3	Hints for External Device Connection.....	129
Chapter 11	Application Linking	131
11.1	Browser Linking.....	132
11.1.1	Browser Launching of i-applis	132
11.1.2	i-appli Launching of Browsers	134
11.2	Mail Linking Launch.....	135
11.2.1	Mail Launching of i-applis	135
11.3	External Device Linking Launch.....	137
11.3.1	External-Device Launching of i-applis.....	137
11.4	In i-appli Linked Launch	141
11.4.1	i-appli Linked Launch Using the Link Mode	141
11.4.2	i-appli Linked Launch with the Launcher Mode	142
11.5	i-appli Update Functionality Linked Operations	144
11.6	Calling Voice-Call Functionality	145
11.6.1	Dialing from i-appli	145
11.6.2	Referencing Individual Identification Information from i-appli.....	145
11.7	Calling Phonebook Management Functionality	146
11.7.1	Adding New Phonebook Groups	146
11.7.2	Recording New Phonebook Entries.....	146
11.8	Calling Bookmark Management Functionality	148
11.9	Calling Schedule Management Functionality.....	148
11.10	Calling Image Data Management Functionality.....	149
11.10.1	Adding New Image Data	149
11.10.2	Image Data Selection Loading	150
11.10.3	Specified Image Data ID Loading.....	150
11.11	Calling Camera Functionality.....	151
11.11.1	Camera Control	151
11.11.2	Exchanging Photographed Images Between Mobile Phones	152
Chapter 12	The Infrared Remote Control	154
12.1	Control signal Structure.....	154
12.2	The Infrared Remote Control API	156
12.2.1	IrRemoteControl.....	157
12.2.2	IrRemoteControlFrame.....	158
Chapter 13	Building, Testing, and Packaging the i-appli.....	159
13.1	Installing the i-appli Development Environment.....	159
13.2	Components of the i-appli Development Environment	159
13.3	Overview of the Development Cycle	161
13.4	Building the Client-Server Application	162
13.5	Packaging the i-appli	162
13.5.1	Creating the ADF	162
13.5.2	Creating the JAR File	167

13.6	Testing the i-appli	168
Chapter 14	Distributing the i-appli	169
14.1	Configuring the Web Server for Serving the i-appli	169
14.2	Configuring the Web Server for Server-side i-appli	170
14.3	Manufacturer-specific i-appli Downloads	170
14.4	Launching i-appli Immediately After Download	171
Appendix A	FOMA Mobile Phone i-appli Runtime Environment—FOMA03 Latest Specification Models.	173
A.1	i-appli Runtime Environment Optimization for FOMA	173
A.1.1	Network Architecture	173
A.1.2	Application Size and Network Communication Size	174
A.1.3	Media Data	174
A.1.4	Application Lifecycle	175
A.2	Additions/Changes to APIs in DoJa-3.5 Profile	176
A.2.1	Transition from i-appli Option APIs to i-appli Standard APIs	176
A.2.2	Newly Added APIs.....	177
Glossary		178

Purpose of This Document

The i-appli Content Developer's Guide describes information needed for i-Mode content providers to design and develop i-appli applications that provide new Java-based services for the i-Mode mobile phone. The guide's objective is to provide a simple model for creating and deploying an i-appli application. It also describes the technology and details of the process that an application developer should follow in order to create usable i-appli applications. This preface provides an overview to the Developer's Guide and contains the following topics:

- Target Audience
- How to Use this Guide
- Typographical Conventions
- Related Documents

Target Audience

The i-appli Content Developer's Guide provides information for developers of services which use i-appli. You should read this guide if you are a service designer, a server or client program programmer, or responsible for application development. The guide assumes that you are familiar with the following:

- Basic knowledge of network applications using the Java programming language
- The product specifications for the i-Mode phone(s) on which the i-appli application will be deployed

How to Use this Guide

The following table shows where to find specific information.

If you are looking for:	See:
i-appli Runtime Environment.	Chapters 1 – 2
Design considerations.	Chapter 3
Guide to specifications.	Chapters 4 – 12
Development process.	Chapters 13 – 14

Typographical Conventions

The following table describes the typographical conventions this guide uses.

Convention	Description
<code>Courier</code>	Courier text denotes code examples, classes, interfaces, methods, and data types
Bold text	Bold text denotes menu and softkey options..
[]	In explanations of syntax etc., items surrounded by brackets ([]) indicate options. In actual designations, these can be set or omitted as needed.
< >	In explanations of syntax etc., items surrounded by angle brackets (< >) indicate places that would be replaced by setting the appropriate value.

In this guide, considerations specific to the DoJa-2.0 and DoJa-3.0 profiles are indicated with [DoJa-2.0] and [DoJa-3.0]. Unless otherwise noted, considerations specific to the older profiles are applicable in the new profiles.

Related Documents

Documents related to this guide are shown below.

- i-appli Content Developer's Guide for DoJa-3.0: API Reference
- i-appli Content Development Guide for DoJa-3.0: Trusted i-appli
- i-appli Content Developer's Guide for DoJa-3.0: i-appli Options and Extensions
- i-appli Content Developer's Guide for DoJa-3.0: Option and Extension API Reference
- i-appli Development Kit for DoJa 3.0 APIs User's Guide
- Java 2 Platform, Micro Edition (J2ME) Connected Limited Device Configuration (CLDC) specification (<http://java.sun.com/products/cldc/>)

Note that any URLs included above are subject to change without notice.

Chapter 1

Introduction



An i-appli-ready mobile phone combines telephone access, Internet access, and the ability to download and run Java applications (i-appli) from any i-Mode content provider's Web site. This document provides programming guidelines for software engineers developing i-appli.

The i-appli mobile phone supports the Connected, Limited Device Configuration (CLDC) of Java 2 Platform, Micro Edition (J2ME). The J2ME CLDC defines a minimum required complement of Java virtual machine specifications and class libraries for small network-connected devices. The CLDC is based on the K Virtual Machine (KVM). The KVM is a compact, highly portable Java virtual machine specifically designed to run on memory-/CPU-/power-constrained devices like the mobile phone (unless specifically noted, the term "mobile phone" is used in this document to refer to the i-appli-ready mobile phone). As a member of the Java 2 family, KVM shares much of the infrastructure of the larger editions of the Java runtime environment; but in order to meet the requirements of small devices, it is optimized for these devices' constraints. The J2ME/CLDC platform, though compact, still provides the Java virtual machine, associated libraries, and APIs DoCoMo content providers need in order to make rich, exciting i-applis that the i-Mode user can install on demand.

The write-once, run-anywhere aspect of Java means that you, the content provider, can write your i-appli once and easily run it on any mobile phone from any manufacturer, regardless of CPU or operating system. Java provides a method for transferring applications across the Internet in secure ways, so that i-Mode users can download, and install i-applis whenever they want, just by connecting to an i-Mode content provider's Web site. Using Java, i-Mode content developers can develop a wide range of applications, including games and robust e-commerce services. Implementing Java in i-Mode mobile phone allows you to deploy newer, more dynamic services.

The remainder of this chapter describes the features and capabilities of the i-appli mobile phone application runtime environment (the i-appli Runtime Environment).

1.1 Benefits of the i-appli Runtime Environment

As a Java content provider, you will write i-appli in the Java programming language. The i-appli Runtime Environment provides the APIs (or i-appli APIs) that you will use to control the mobile phone's user interface, communications, text-conversion, graphics, multimedia, and storage of data to the mobile phone. The following three groups of APIs make up the i-appli API set.

- i-appli standard API

These APIs and their operations are designated with common specifications and are installed as standard on all models.

- i-appli optional API

Although these APIs and their operations are designated with common specifications, it is the responsibility of the manufacturer to decide whether or not they will be installed. When using this category of API, it must be presumed that they will not be installed on certain models.

- i-appli extension API

Even in cases where identical functions are being realized, there is a possibility that differences may exist in these APIs and their operations depending on the manufacturer in question. When using this category of API, it must be presumed that they will not be installed on certain models, and also that usage methods may vary from model to model. (Even in the case of models with identical functionality, there is no guarantee of compatibility at the source code level.)

[DoJa-2.0]

The classification of API categories for the i-appli Runtime Environment differs from that for DoJa-1.0 Profile. Details regarding all APIs from the above-mentioned three categories are published in the i-appli Content Developer's Guide for DoJa-2.0 Profile or later.

Note that this document, i-appli Content Developer's Guide deals mainly with descriptions of J2ME/CLDC APIs and the i-appli standard APIs. For specifics concerning the i-appli optional and extension APIs, please refer to the "i-appli Content Development Guide: i-appli Option and Extension" and the "i-appli Content Development Guide: i-appli Option and Extension API Reference".

In addition, the i-appli Runtime Environment provides the following capabilities

- Dynamic delivery of i-applis
- The Java Application Manager (JAM)
- Security features, including the "simple sandbox model," which specifies that only a limited set of "safe" Java APIs, such as the classes in `java.lang.*` and `java.util.*` packages are available to the developer.

The i-appli Runtime Environment consists of four layers of APIs.

- APIs defined by J2ME/CLDC
- i-appli APIs (i.e., i-appli standard APIs)
- i-appli APIs (i.e., i-appli optional APIs)
- i-appli APIs (i.e., i-appli extension APIs)

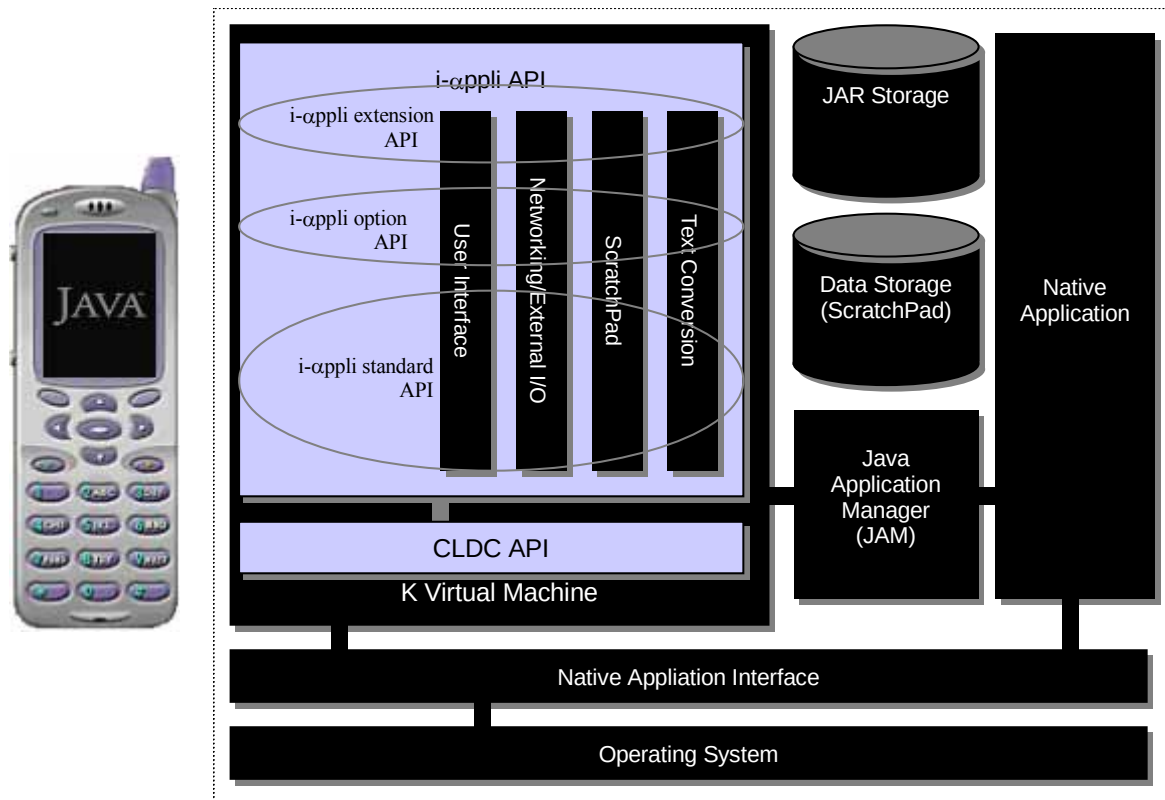


Figure 1: The i-appli Runtime Environment in an i-appli compatible mobile phone

Notes:

- Most i-applis will use only the J2ME/CLDC and the i-appli standard API. Some specialized i-applis may require improved expressive capabilities, and these applications will make use of the i-appli optional APIs and i-appli extension APIs. Please be aware that, depending on the manufacturer of the phone in question, it may be impossible to execute i-applis which use these APIs.
- The i-appli optional and extension APIs include items which may be adopted in the i-appli standard APIs of future profiles. Conversely, it also includes items which may be removed in future profiles.

The J2ME/CLDC defines a standard Java platform for small, resource-constrained, network-connected devices that are characterized as follows:

- A small amount of total memory.
- Limited power – often battery operation.
- Connectivity to some kind of network – often with a wireless, intermittent connect and with limited bandwidth..
- Need for a user interface optimized for the service provided by devices.

Mobile phones, two-way pagers, personal digital assistants, home appliances, and point-of-sale terminals are typical devices that are supported by the CLDC specification.

The J2ME/CLDC represents a stable application platform installed into the i-appli Runtime Environment. Like the larger Java editions, J2ME aims at maintaining the qualities for which Java technology has become known:

- Built-in consistency across products in terms of running anywhere, any time, on any device.
- High-level object-oriented programming language with a large developer base and development tools.
- Portability of code
- Safe network application delivery
- Upward scalability with J2SE and J2EE.

The i-appli Runtime Environment for CLDC includes APIs with the following types of functionality:

- Network connectivity using HTTP and HTTPS.
- Components for defining user interface elements
- Low-level graphics controls
- A ScratchPad used for storing data.
- Stand-by application control which makes i-applis resident on mobile phones in the stand-by condition
- Linked operation of i-appli and other applications (browsers, mailers, i-appli, etc.)
- External connection using the infra-red port (IrOBEX, or infrared remote control)

[DoJa-2.0]

Stand-by applications, IrOBEX external connection, and native application linking represent functions which are newly installed with DoJa-2.0 Profile. Detailed descriptions of these functions are provided in Chapter 9, Chapter 10, and Chapter 11 respectively.

[DoJa-3.0]

The infrared remote control feature and application linking between i-appli are newly introduced features of DoJa-3.0 Profile. The infrared remote control feature will be explained in detail in Chapter 12.

1.1.1 Optimized Support for Standalone and Client-server Applications

i-appli services support programming of both stand-alone and client/server configurations.

Standalone i-applis do not have to interact with a server, because both the application and its data reside on the mobile phone. Many games, calculators, and utilities can be written as standalone i-applis.

Standalone i-applis do not require a server connection to perform the tasks for which they were designed; however, a server connection is required to download an i-appli to the mobile phone. Chapter 14 discusses issues related to making your i-appli available to mobile phones over the Internet.

While standalone i-applis can provide value, the most interesting i-applis will almost certainly be client-server. Client-server extend the limited processing power of a mobile phone with the vastly greater processing power, and the powerful data retrieval capability of the server. The i-appli Runtime Environment includes an API which performs network connections based on HTTP or HTTPS to connect the i-appli with the content provider's server through the internet. It also includes an IrOBEX API for operating 2 physically close cellular phones as clients and/or servers via the infrared port.

HTTP(S) Connection

With i-applis which use HTTP(S) connection, the processing capabilities of the client computer (in this case, a mobile phone) can be combined with the processing capabilities of a remote internet server or with data access functionality. Mobile phones are limited in terms of processing power and memory capacity; however, they can function as the front-end processors for servers with extensive memory and powerful processing capabilities. Although restrictions also apply with regard to a mobile phone's data input methods and display size, the creation of i-applis based on optimum design and programming allows the users of i-appli-capable mobile phones to conveniently access information and remote services such as electronic commerce.

The typical role of the server for client-server type i-applis is to search for the required data in databases and to then return this data to the requesting client. In certain cases, however, where this is unsuitable or impossible as a result of restrictions in the client's processing power or memory capacity, the server may also carry out a portion of the calculation processing.

For more details regarding the creation of client-server i-applis using HTTP(S) connection, refer to Chapter 5.

IrOBEX External Connection

IrOBEX external connection (hereinafter referred to simply as OBEX) realizes a simple client-server model where the infra-red ports integrated into mobile phones are used to carry out the communication of data over short distances between these phones. Containing both client and server API for OBEX external connection, the i-appli Runtime Environment provides support for the communication of data between mobile phones. This functionality can be used to realize business-card exchange applications, schedule exchange applications, and other similar i-applis which transfer small amounts of data between mobile phones.

The hardware specifications of the connecting device and the surrounding environment may make it impossible to carry out the communication of data using OBEX external connection.

Note that when not expressly noted in the following section, the terms “client” and “server” will refer to situations where both the client and server can communicate using HTTP(S). For OBEX, these will be referred to as the OBEX client and the OBEX server.

1.1.2 Support for Local Storage of Persistent Data

The i-appli Runtime Environment provides for a local data storage mechanism called ScratchPad, which secures storage space on the mobile phone for the storage of persistent data. Standalone i-applis that need to store persistent data will do so using the ScratchPad, while client-server i-applis will typically store only some data using the ScratchPad, and store the rest on the server side.

On the server side, persistent data is almost always stored in databases. Client data is sent to the server via an HTTP request, which interacts with a servlet or CGI script. The servlet or CGI script then acts as a middle-man between the HTTP request and the database access protocols. This means there will always be an HTTP server on the server side.

Consider the nature of the persistent data before you decide where to store it. Store persistent data using the ScratchPad only if the user can afford to lose it; otherwise, store it on the server side. This is because data stored using the ScratchPad could be lost due to hardware failure or other problems in the mobile phone.

One of the security features of i-appli mobile phone is that ScratchPad memory allocated to an i-appli by the Java Application Manager (JAM: See 1.2) cannot be accessed by another i-appli. In addition, only one i-appli can run on a mobile phone at any one time, preventing multiple i-applis from sharing data in application execution memory. If you want to share data between i-applis, you will need to share the same data storage area for each i-appli on the server side.

1.2 Java Application Manager

A mobile phone implementing the i-appli Runtime Environment provides a component for managing the Java i-applis installed on that mobile phone. This component is called the Java Application Manager (JAM), and it is responsible for

- Displaying existing i-applis stored on the mobile phone.
- Managing the execution of i-applis (i.e., startup and forced shut-down, mediation between the i-appli Runtime Environment and applications, etc.)
- Installing and updating i-applis.
- Deleting i-applis stored in the mobile phone. Deleting an i-appli saved on the cellular phone is performed with an explicit instruction from the user.

JAM is an independent native component from the KVM, and cannot be controlled from the i-appli program logic (hereafter called the application program).

Should an i-appli fail to terminate after the user quits out of it, the user can instruct JAM to force termination by pressing the mobile phone's Hang-Up key.

[DoJa-2.0]

With DoJa-2.0 Profile or later, the forced termination of i-applis is incorporated into the Hang-Up key. Note that with DoJa-1.0 Profile, key assignments for forced termination depend on the manufacturer in question.

1.3 The i-appli Mobile Phone Java Security Model

In addition to the security features inherent to CLDC, the Java Application Manager (JAM) provides the following security features specifically for the i-appli mobile phone execution environment:

- Only one i-appli can run at a time on the mobile phone. This ensures that one i-appli cannot interfere with or access data from another. Nevertheless, KVM supports multi-threads, thus allowing multi-thread programming where communication and user-interface processing can both be carried out in parallel within a single i-appli.
- JAM is a native component that is independent from the KVM, and cannot be controlled from an i-appli. The JAM manages the downloading of i-applis. The result is that the mobile phone protects the user from abusive or buggy behavior of downloaded i-applis.
- i-applis only have limited access to storage on the mobile phone (the ScratchPad). This means that no i-appli is authorized to access the ScratchPad memory allocated to another i-appli. Therefore, the downloaded i-appli cannot perform improper actions, such as reading personal information stored in ScratchPad by another i-appli and sending it to a different server or rewriting it without the user's knowledge.
- It is not possible for i-applis to access native data which includes memory-dial data and other personal information. Furthermore, in order to prevent web browsing and voice-call transmission against the wishes of the user, the confirmation of the user will always be sought upon API calling for functions which access other applications on the mobile phone such as browsers and dialers. This allows the user to safely download and execute i-applis.

[DoJa-2.0]

As a result of security issues, no support for linked operation between i-applis and applications was provided with DoJa-1.0 Profile. In the case of DoJa-2.0 or later, whenever linked operation is to be carried out between these types of applications, a process is included which will always seek the user's approval, thus ensuring support for safe application linking by users.

- The i-Mode Java Application Environment communication function provides a simple, easy-to-use interface for both Java-compatible i-Mode supported HTTP and HTTPS communication. i-applis can communicate only with the server from which they were downloaded. This network security feature protects the user from an i-appli sending information to another server where the user does not expect it to go.

[DoJa-3.0]

In DoJa-3.0 Profile, only authorized i-applis adopt the relaxed security features of the trusted i-appli framework (service name "i-appli DX"). Trusted i-appli can operate under loosened security conditions, and thus are able to access native data, such as the phonebook, connect to servers other than the i-appli's download source, etc. However, in trusted i-appli too, access is subject to constant restrictions with regards to phonebook data for specific individuals to keep it from being sent outside the cellular-phone.

These features (trusted API) can only be used in the services listed on the i-mode menu registered on the iMode server. The functionality of the trusted API cannot be used by other i-applis than trusted i-applis.

The details of trusted i-appli will not be covered in this guide.

1.4 i-appli Installation and Lifecycle

The only way to run an i-appli on a mobile phone is to download (install) it from a content provider's i-Mode compatible Web site. When an i-appli is downloaded, JAM allocates enough of each type of memory to store the i-appli. Both the i-appli JAR file and the ScratchPad are stored in the mobile phone's non-volatile memory space.

The following subsections describe the i-appli lifecycle: how an application is downloaded, executed, and deleted.

1.4.1 Downloading the i-appli

The general procedure for downloading i-applis is given below. Note that some details of how a given mobile phone will carry out a given action are manufacturer-specific.

1. The user locates the desired i-appli's download link using the i-Mode browser.

i-appli downloads are started by operating the i-Mode browser. The Web page (HTML content) for the download typically displays a description of the i-appli and a link that the user clicks to start the download process. The link contains a reference to an application descriptor file (a file with a ".jam" extension; also called an ADF or JAM file. In this document, they will be called ADFs). The ADF is a text file in SJIS code, and records information about the i-appli associated with that ADF with one key and value pair on each line.

One purpose of this file is to allow the JAM to decide whether the selected i-appli can be successfully downloaded to the mobile phone. If the JAM is able to decide beforehand that the i-appli cannot be successfully downloaded, it can save the user the cost of actually downloading the i-appli to the mobile phone. The ADF is small (around 100 – 300 bytes), while an i-appli is several k bytes – several tens of k bytes. So it is advantageous in terms of communication costs to download the ADF and check for available space and compatibility before downloading the entire i-appli.

Section 13.5.1 details the contents of the ADF, and Chapter 14 describes the Web server standards for delivering i-applis to mobile phones.

2. The user clicks the link to start the installation process.

This user operation makes the browser pass the URL showing the location of the ADF to the JAM. After this, the JAM handles the downloading and installation of the i-appli.

3. The JAM verifies the content of the ADF, and confirms whether to install the i-appli.

The JAM makes sure that all the mandatory keys are present in the ADF, then verifies whether it is possible to install the i-appli on that mobile phone, from the content of the ADF (the JAR file size, ScratchPad size, target mobile phones, etc.). If installation is possible, it then proceeds to the next step.

Note that in the following situations, the i-appli will be deemed to have already been downloaded to the mobile phone and installation will not be possible. (i-applis shown in section 1.4.3 can be updated).

- The mobile phone already contains an installed i-appli whose ADF contains an identical URL to that of the i-appli being installed. (In DoJa-3.0 Profile, when this sort of i-appli download is instructed, the cellular-phone checks to see if that i-appli can be updated. If it can, update handling of the i-appli occurs as described in section 1.4.3.)
- Cases in which an i-appli is already installed on the cellular phone with a different URL ADF than the i-appli to be installed, but having the same name (ADF AppName key) and JAR file URL (ADF PackageURL key).

4. The JAM downloads the i-appli.

If the JAM decides that the i-appli can be installed on the mobile phone, it obtains the JAR file's URL using the `PackageURL` key (if the `PackageURL` key is a relative URL, the ADF location is used as the base URL), and starts the HTTP download process. If the user interrupts the download process, or a communication error occurs, the JAM discards the partially downloaded i-appli, leaving the mobile phone as if the i-appli was never downloaded.

5. The JAM updates the list of installed i-applis.

The JAM adds the i-appli to the list of i-applis installed by the process explained above and stores the ADF information and the JAR file in nonvolatile memory.

If the ADF specifies that the i-appli will use features which require user permission, such as HTTP communications, the user will be asked to grant permission for i-appli to use these features here. Furthermore, if the ADF settings are such that i-applis can operate as a stand-by application, then the user will be asked whether or not the application is to be registered as such.

The i-appli will be installed into the mobile phone during the above steps 1 through 5, and it will then be ready for use (i.e., execution).

[DoJa-3.0]

In DoJa-3.0 Profile, launch functionality immediately after download is supported so that downloading and launching an i-appli can be performed with a single click from the user (this sort of i-appli is referred to as an i-appli that launches immediately after download). Because i-appli that launch immediately after download require no user confirmation, they are limited to using functionality for which basic user confirmation is necessary.

For details regarding i-appli that launch immediately after download, see chapter 14.

1.4.2 Launching the i-appli

The procedure for launching an i-appli and the user interface are manufacturer-specific. A typical scenario follows:

1. A list of installed i-applis is displayed in response to a user command.
2. The user selects the i-appli they wish to launch from the list.
3. The JAM launches the Java Virtual Machine specifying the "main" class of the i-appli, as indicated by the ADF's `AppClass` key.
4. The Java i-appli's screen appears on the mobile phone's display.

Furthermore, the above-mentioned user operations do not result only in launching of the i-appli; rather, the i-appli Runtime Environment supports the following variations in startup condition.

- Timer startup
- i-appli startup from a browser, mailer, or external device (i.e., infra-red port), i-appli through linking with an i-appli.
- Automatic startup following return to the stand-by condition through registration as the stand-by application.
- Launch immediately after download

For more details regarding timer startup, refer to the item `LaunchAt` Key from Section 13.5.1. Similarly, for more details regarding application linking or stand-by applications, refer to Chapter 11 and Chapter 9 respectively. Regarding i-appli that launch immediately after download, see chapter 14.

[DoJa-2.0] [DoJa-3.0]

Only user-operated startup and timer startup were supported in DoJa-1.0 Profile. From DoJa-2.0 Profile on, i-appli launching from browsers, mailers and external devices via application linking and launching of stand-by applications in stand-by were added. Further, launching i-appli from other i-appli and launching immediately after download were added in DoJa-3.0 Profile.

1.4.3 Updating an i-appli

i-applis are updated when the user gives an i-appli update command, for instance by making a menu selection on a list of installed i-applis (the exact procedure is manufacturer-specific).

When the JAM installs an i-appli, it stores the URL of the ADF. When the user gives a command to update an i-appli, the JAM accesses this URL again, and obtains a new ADF. Then, the JAM decides whether the i-appli needs to be updated, based on the content of this ADF. It only begins the update process if it decides that an update is actually necessary.

To update an i-appli (for example, to fix bugs or add new features), the content provider must do the following:

- Set the `LastModified` key in the ADF to a later date than the previous version. Configure other keys to appropriate values as necessary. However, please do not set the `ScratchPad` (the `SPsize` key) size of the application to be updated to a smaller value than the previous version (When managing `ScratchPad` segmentation, the size of individual segments should not be smaller than those in the previous version). As long as the size of the updated i-appli's `ScratchPad` is at least as large as that of the previous version, the updated i-appli will inherit the contents of the previous `ScratchPad`.
- Post the updated JAR file and ADF file on the content provider's Web site. The name of the new ADF is the same as that of the old one. Please be aware that, if the ADF's URL is changed, a user having already downloaded an i-appli to their mobile phone will be unable to update this i-appli.

Even if the user gives a command to update an i-appli, the JAM will not update it if the `LastModified` key of the ADF has not been updated.

In DoJa-3.0 Profile besides i-appli updates performed by the user, functionality has been added so that using the API, the i-appli can encourage the user to update the i-appli. This functionality is achieved through one kind of application linking, in which the native component for updating i-appli is called from i-appli. See Chapter 11 regarding the i-appli update API.

[DoJa-3.0]

For models of phone without the i-appli update API (DoJa-2.0 Profile or earlier), it will be necessary to notify the user about updates through a Web site or mail.

1.4.4 Deleting the i-appli

Each manufacturer has a procedure for removing i-applis at the request of the user. When the JAM receives a command from the user to delete an i-appli, it deletes the JAR file, and the `ScratchPad` allocated to the i-appli, from the mobile phone.

Chapter 2

i- α pli Runtime Environment

This chapter describes the makeup of the functions and APIs of the Java Application Runtime Environment (i.e., the i- α pli Runtime Environment) implemented on i- α pli-capable mobile phones.

2.1 i- α pli Runtime Environment

The i- α pli Runtime Environment is comprised of the mobile phone's native applications, the Java Application Manager, the i- α pli API, the K virtual machine, and the CLDC class libraries. The following diagram shows the makeup of the i- α pli Runtime Environment.

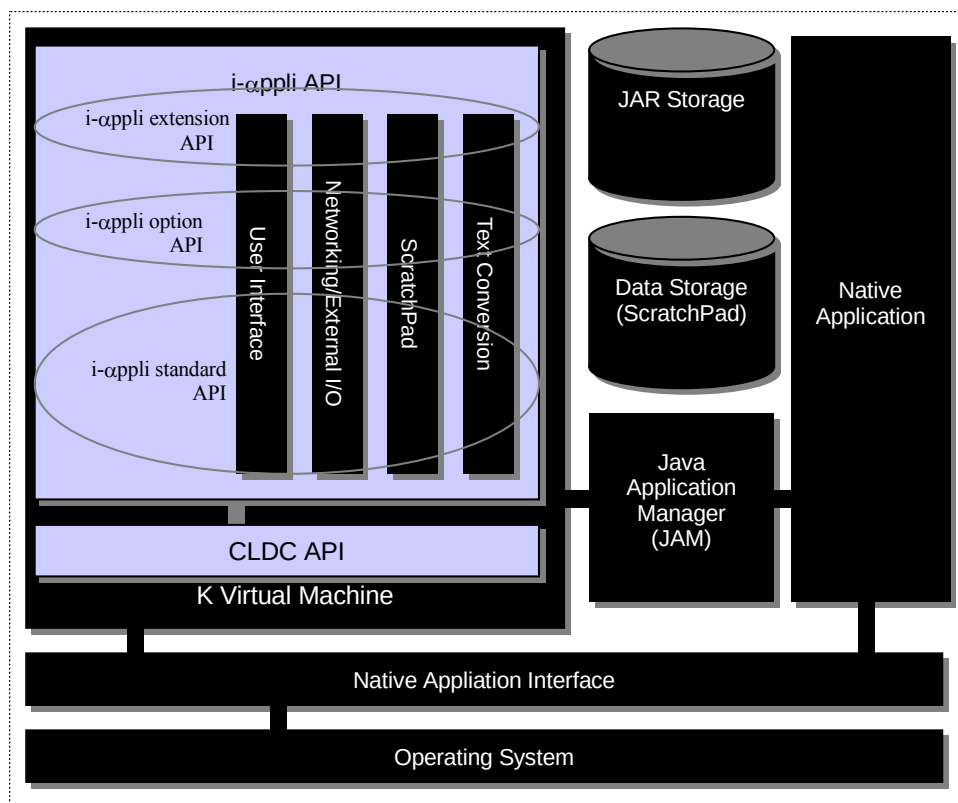


Figure 2: i- α pli Runtime Environment

2.1.1 Native Applications and Operating System

Ordinary telephones and traditional browser-based i-Mode services use the operating system and native (e.g. the Browser, networking components, and Dialer) on the mobile phone to operate. The i-appli Runtime Environment also includes functionality for the linked operation of these native applications using the JAM.

2.1.2 Java Application Manager

The Java Application Manager (JAM) manages individual i-applis stored as JAR files, and communicates with the K virtual machine.

The main features of the Java Application Manager are shown in Section 1.2.

2.1.3 K Virtual Machine

The K virtual machine is a version of a Java virtual machine that was re-designed for small embedded devices. It is designed to run on memory-/CPU-/power-constrained devices.

2.1.4 CLDC API

The CLDC APIs include a subset of classes inherited from Java 2 Standard Edition (J2SE). The CLDC APIs share many of the features included in the larger editions of the Java runtime environment, but in order to meet the requirements of small devices, it is optimized for their constraints.

The CLDC APIs are shown in Section 2.2.2.

2.1.5 i-appli API

The i-appli APIs are designed for i-appli-capable mobile phones. While the CLDC provides the application infrastructure that can be shared on small embedded devices, the i-appli APIs build on the CLDC base offer application infrastructures tailored to i-appli services. The following individual elements comprise the i-appli APIs:

- User Interface APIs
- Networking APIs
- SJIS text processing
- ScratchPad local storage
- Application linked operation API and native functionality call API
- Infrared port control (External-connection and infrared remote control via OBEX)

Refer to Section 2.2.3 for more details regarding the i-appli APIs. Chapters 4 and later discuss the exact procedures for using each feature in detail.

Note that the i-appli APIs are separated into the following three categories:

- i-appli standard APIs: These APIs and their operations are designated with common specifications and are installed as standard on all models.
- i-appli optional APIs: Although these APIs and their operations are designated with common specifications, it is the responsibility of the manufacturer to decide whether or not they will be installed.
- i-appli extension APIs: Even in cases where identical functions are being realized, there is a possibility that differences may exist in these APIs and their operations depending on the manufacturer in question.

Please ensure that neither i-appli optional APIs nor i-appli extension APIs are used for i-applis which are required to operate on all models.

2.2 Overview of API in the i-appli Runtime Environment

The remainder of this chapter describes the APIs supported in i-appli Runtime Environment. This section is subdivided as follows:

- 2.2.1 Network Architecture
- 2.2.2 CLDC APIs
- 2.2.3 i-appli APIs

The CLDC APIs contain the subset class from the standard J2SE library, and although mapping to J2SE is functionally possible, these APIs also contain a CLDC-specific class which has been redesigned for small devices. This issue was settled through the use of Java Community Process (JCP).

Meanwhile, the i-appli APIs have been tailored for imode services, and these are controlled by NTT DoCoMo.

Note that of the i-appli APIs, this section will deal only with the i-appli standard APIs. For specifics concerning the i-appli optional and extension APIs, please refer to the “i-appli Content Development Guide: i-appli Option and Extension” and the “i-appli Content Development Guide: i-appli Option and Extension API Reference”.

2.2.1 Network Architecture

The i-appli network architecture is nearly identical to the network architecture of conventional i-Mode services. Application-level communication between a mobile phone and a content provider's site is conducted using HTTP. Below is a conceptual diagram of the network architecture:

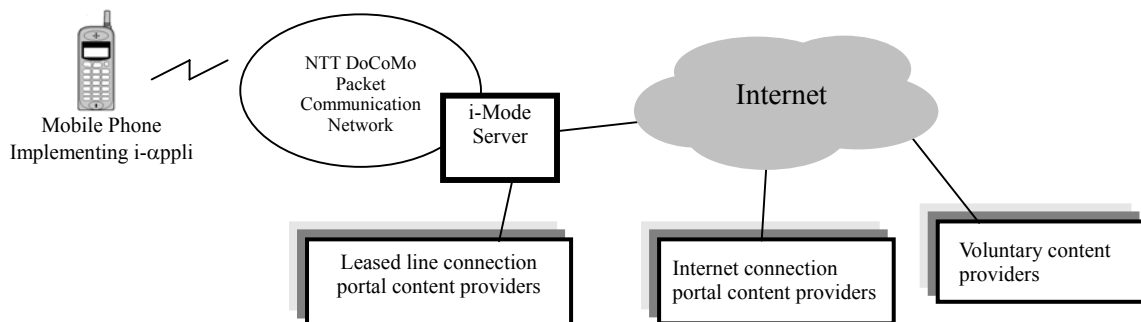


Figure 3: Conceptual Diagram of the Network Architecture

i-appli service network operations can be divided into the following two categories:

- Downloading and installing i-applis
- Accessing a server while the i-appli is running

As with conventional i-Mode services, i-appli service network operations are conducted via an i-Mode server. Even on the mobile phone, HTTP is used for the application-level communication protocol, and so the i-Mode server does not conduct any processing along the lines of protocol conversion. When i-applis are downloaded as well, the file is sent from the content provider-side Web server to the mobile phone as-is.

Additionally, with i-appli service, server authenticated is supported by the secure socket layer (SSL) on the mobile phone side. By using the communication capabilities and SSL support of the i-appli service, it is possible to conduct secure HTTP (HTTPS). In this case, the secure interval is from the mobile phone to the content provider-side Web server.

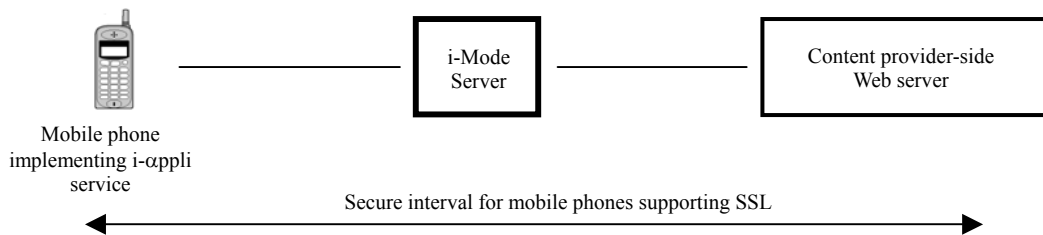


Figure 4: SSL Support of i-Mode Services

Note that this service does not support client authentication, where a certificate is passed to the mobile phone (client) side.

2.2.2 CLDC API

The majority of the class libraries included in CLDC are a subset of the larger editions of the Java execution environment (J2SE) to ensure upward compatibility and portability of Java applications. While upward compatibility is a very desirable goal, J2SE libraries have strong internal dependencies that make it difficult to create subsets in such important areas as security, input/output, User Interface, networking and storage. These dependencies are a natural consequence of the design evolution and reuse that have occurred as the Java libraries have developed over time. Unfortunately, these dependencies make it very difficult to take just one part of the libraries without including several others. For this reason, some of the libraries have been redesigned, especially in the areas of networking and I/O.

J2SE Classes Supported in CLDC

CLDC provides a large number of classes inherited from Java 2 Standard Edition (J2SE). The functions of these classes and the modes of use for inherited methods are not changed. The classes do not add any public or protected methods or fields that are not available in the corresponding J2SE classes. This is because the rules for J2ME configurations mandate that each class that has the same name and package name as a J2SE class, must be a strict subset of the corresponding J2SE class, and do not allow independent methods and fields to be added.

System classes

Standard Java class libraries include several classes that are intimately linked with the Java virtual machine. Additionally, several standard Java tools assume the presence of particular classes in the system. For instance, the standard Java compiler generates code that requires particular methods of classes `String` and `Stringbuffer` to be available. The following system classes are included:

```

java.lang.Object
java.lang.Class
java.lang.Runtime
java.lang.System
java.lang.Thread
java.lang.Runnable (interface)
java.lang.String
java.lang.StringBuffer
java.lang.Throwable
  
```

Data type classes

CLDC supports the following basic data type classes from package `java.lang`. Each of these classes is a strict subset of the corresponding class in Java 2 Standard Edition. Because CLDC does not support floating point numbers, it does not support basic data type classes that express floating point type data:

```
java.lang.Boolean
java.lang.Byte
java.lang.Short
java.lang.Integer
java.lang.Long
java.lang.Character
```

Collection classes

CLDC supports the following Collection classes from package `java.util`.

```
java.util.Vector
java.util.Stack
java.util.Hashtable
java.util.Enumeration (interface)
```

I/O classes

CLDC supports the following classes from package `java.io`. Classes `Reader`, `Writer`, `InputStreamReader`, and `OutputStreamWriter` are required in order to support internationalization:

```
java.io.InputStream
java.io.OutputStream
java.io.PrintStream
java.io.ByteArrayInputStream
java.io.ByteArrayOutputStream
java.io.DataInput (interface)
java.io.DataOutput (interface)
java.io.DataInputStream
java.io.DataOutputStream
java.io.Reader
java.io.Writer
java.io.InputStreamReader
java.io.OutputStreamWriter
```

Micro Edition classes

CLDC supports the following classes from package `javax.microedition.io`.

```
javax.microedition.io.Connector
javax.microedition.io.ConnectionNotFoundException
javax.microedition.io.Connection (interface)
javax.microedition.io.ContentConnection (interface)
javax.microedition.io.Datagram (interface)
javax.microedition.io.DatagramConnection (interface)
javax.microedition.io.InputConnection (interface)
javax.microedition.io.OutputConnection (interface)
javax.microedition.io.StreamConnection (interface)
javax.microedition.io.StreamConnectionNotifier (interface)
```

Calendar and time classes

CLDC includes a small subset of the standard J2SE classes `java.util.Calendar`, `java.util.Date`, and `java.util.TimeZone`. CLDC supports GMT and UTC time zones. Additional time zones may be provided by actual implementations. The i-appli Runtime Environment supports JST as an additional time zone:

```
java.util.Calendar
java.util.Date
java.util.TimeZone
```

Additional utility classes

CLDC provides two additional utility classes. Class `java.util.Random` provides a simple pseudo-random number generator that is useful for implementing Java applications such as games. Class `java.lang.Math` provides methods `min()`, `max()`, and `abs()` (for data types `int` and `long`) that are frequently used by other Java library classes:

```
java.lang.Math
java.util.Random
```

Error and Exception classes

Since the class libraries included in CLDC are generally intended to be highly compatible with J2SE libraries, the classes included in CLDC throw precisely the same exceptions as their corresponding J2SE classes do. Consequently, the CLDC class libraries include a large number of exception classes:

Error classes

```
java.lang.Error
java.lang.VirtualMachineError
java.lang.OutOfMemoryError
```

Exception classes

```
java.lang.Exception
java.lang.ClassNotFoundException
java.lang.IllegalAccessException
java.lang.InstantiationException
java.lang.InterruptedException
java.lang.RuntimeException
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.lang.ClassCastException
java.lang.IllegalArgumentException
java.lang.IllegalThreadStateException
java.lang.NumberFormatException
java.lang.IllegalMonitorStateException
java.lang.IndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException
java.lang.StringIndexOutOfBoundsException
java.lang.NegativeArraySizeException
java.lang.NullPointerException
java.lang.SecurityException
java.util.EmptyStackException
java.util.NoSuchElementException
java.io.EOFException
java.io.IOException
java.io.InterruptedIOException
java.io.UnsupportedEncodingException
java.io.UTFDataFormatException
```

JVM Features Not Supported on CLDC Platform

The following features have been eliminated from the standard Java virtual machine to provide a smaller footprint for the CLDC virtual machine (KVM).

Floating-point support

The KVM does not support floating-point operation.

JNI

The KVM does not support the Java Native Interface (JNI).

Thread Groups

The KVM implements multithreading, but does not have support for thread groups. Thread operations such as starting and stopping of threads can be applied only to individual thread objects. If application developers want to perform thread operations for groups of threads, they must use explicit collection objects at the application level to manage the thread objects as a group.

Finalization

CLDC does not include the method `finalize()` in the class `Object`, and therefore the KVM does not support finalization of objects. The application does not define the method `finalize()`.

Generic Connection Framework (`javax.microedition.io` package)

The requirement for having a small footprint J2ME implementation has led to the generalization of the J2SE network and I/O classes. The goal for this new framework is to be a precise functional subset of the J2SE network and I/O functions, which can be easily mapped to a common low-level hardware or to any J2SE implementation, but with better extensibility, flexibility, and coherence in supporting new devices and protocols. Instead of providing access methods based on individual abstractions for each type of communication destination, access methods based on an integrated abstraction are used at the application programming level as detailed below.

All connections are created using class method `open()` in the `Connector` class. If successful, the method returns an object implemented in one of the generic connection interfaces (one of the sub-interfaces with the `Connection` interface being the root). This interface group is made up of a large number of interfaces, as shown in Figure 5.

The `open()` method takes a URL parameter in the general form:

```
Connector.open("<protocol>:[target][parameters]");
```

An objective of this design is to isolate, as much as possible, the differences between the setup of one access target and another, into a string characterizing the type of access target (in this case the URL string). The parameter to `Connector.open()` is this URL string. A key benefit of this approach is that the bulk of the application program stays the same regardless of the kind of connection that is used. This differs from the conventional implementation for J2SE and other application runtime environments, in which the abstractions applied in application programs often change dramatically when changing the access target (for instance, when changing from file I/O to socket I/O).

If differences in particular protocol setups can be identified by the `Connector.open()` parameter and absorbed on the platform side, there is no need for protocol-specific processing on the application program side. This can improve application program portability. An application program that gets the parameter string from the runtime environment can be highly portable across both platforms and protocols. This can eliminate or minimize the amount of rewriting required in the application program when changing from one form of connection to another.

The binding of protocols to a application program is done at runtime. At the implementation level, the protocol portion of the URL parameter to `Connector.open()` determines which of the supported protocol implementations is to be used. It is this runtime binding mechanism that allows the program to dynamically adapt to different protocols at runtime. This is similar to the relationship between application programs and device drivers on a PC or workstation.

Connections to different types of devices will sometimes need to conduct operations specific to that type of device. A file, for instance, can be renamed, but no similar operations exist for a TCP/IP. The Generic Connection framework reflects these different capabilities, ensuring that logically identical operations share the same API.

The new framework is implemented using a hierarchy of sub-interfaces inherited from the `Connection` interface that group together classes of protocols with the same semantics. The Generic Connection framework addresses six basic interface types, not counting the `Connection` interface, which is a super-interface for all interfaces:

- Basic serial input: `InputConnection`
- Basic serial output: `OutputConnection`
- Datagram oriented communications: `DatagramConnection`
- Stream oriented communications (sub-interface of `InputConnection` and `OutputConnection`): `StreamConnection`
- A notification mechanism for a server to be informed of client-server connections: `StreamConnectionNotifier`
- Content transmission communications (sub-interface of `StreamConnection`): `ContentConnection`

The collection of `Connection` interfaces forms a hierarchy that becomes progressively more capable as the hierarchy inherits from the root `Connection` interface. This arrangement allows content developers to choose the optimal level of cross-protocol portability for the application program being written. The key inter-class relationships of the Generic Connection class hierarchy are illustrated in Figure 5.

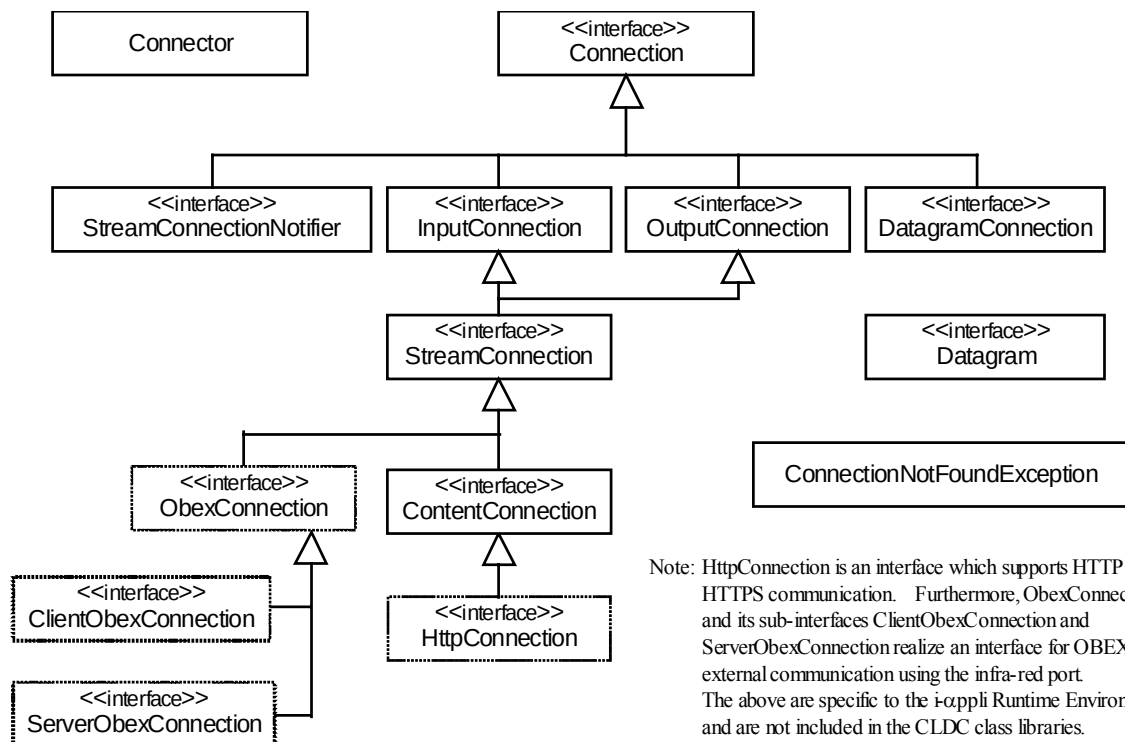


Figure 5: Generic Connection Class Hierarchy

The i-appli Runtime Environment supports the following interfaces for external access by an i-appli:

- HTTP(S) communication: `HttpConnection`
(subclass-interface of `ContentConnection` included in the i-appli API)
- ScratchPad: `StreamConnection`
- Resource: `InputConnection`
- OBEX external connection: `ObexConnection`, `ClientObexConnection`, and `ServerObexConnection` (`StreamConnection` sub-interface as included in the i-appli API)

The following two items are available as optional parameters of the `Connector.open()` method:

- Access mode (`READ`, `WRITE`, `READ_WRITE`)
- Timeout exception request flag

In the access mode, a value is set appropriate to the operation to be accessed. The timeout exception request flag is set to true for HTTP(S) communications and OBEX external-connections, and false for ScratchPad or resource.

[DoJa-2.0]

Although OBEX external connection via the infra-red port is supported for DoJa-2.0 Profile or later, the necessary APIs are also defined in the Generic Connection framework.

2.2.3 i-appli API

The following section will provide a description of the classes for i-appli standard APIs as included in the `io`, `net`, `util`, `lang`, `ui`, `device`, and `system` sub-packages within the `com.nttdocomo` package. (Some of the classes will also contain methods belonging to i-appli optional APIs.)

[DoJa-3.0]

The `com.nttdocomo.device` package and `com.nttdocomo.system` package are new to DoJa-3.0 Profile.

Package `com.nttdocomo.io`

The CLDC Generic Connection framework forms the basis for the i-appli Runtime Environment I/O package. Package `com.nttdocomo.io` provides an interface and exception class for connecting to a network resource specified in URL format, and conducting I/O. This interface supports network connection using the HTTP or HTTPS protocol in addition to OBEX external connection via the infra-red port. This package includes supplementary CLDC input/output classes to efficiently handle text streams.

See Section 2.2.2 for framework of Generic Connection.

The package contains the following interface:

Name	Description
ClientObexConnection	Interface that defines access to the infra-red port in the capacity of an OBEX client. By specifying the URL “obex:/irclient” for the method <code>javax.microedition.io.Connector.open()</code>, an implementation instance for this interface may be obtained. ClientObexConnection is a sub-interface of ObexConnection, the common interface for OBEX servers and OBEX clients within the i-appli Runtime Environment. [DoJa-2.0] This interface is newly added with DoJa-2.0 Profile.
HttpConnection	Interface that defines access to an HTTP connection. An implementation instance of this interface can be obtained by specifying an HTTP or HTTPS URL in the <code>javax.microedition.io.Connector.open()</code> method. HttpConnection. is a sub-interface of the <code>javax.microedition.io.ContentConnection</code> interface of the Generic Connection framework.
ObexConnection	Interface that defines a common interface for OBEX servers and OBEX clients. It is not possible to use directly-implemented instances of this interface in application programs. This interface defines response status and operation codes constant, in addition to common accessor methods for OBEX servers and OBEX clients. ObexConnection is a sub-interface of the Generic Connection framework's <code>javax.microedition.io.StreamConnection</code> interface. [DoJa-2.0] This interface is newly added with DoJa-2.0 Profile.
ServerObexConnection	Interface that defines access to the infra-red port for OBEX servers. By specifying the URL “obex:/irserver” for the <code>javax.microedition.io.Connector.open()</code> method, an implementation instance for this interface may be obtained. ServerObexConnection is a sub-interface of ObexConnection, the common interface for OBEX servers and OBEX clients within the i-appli Runtime Environment. [DoJa-2.0] This interface is newly added with DoJa-2.0 Profile.

The package contains the following class:

Name	Description
<code>BufferedReader</code>	Supports a buffered text input stream. This class offers functionality equivalent to a subset of the J2SE <code>java.io.BufferedReader</code> class. [DoJa-3.0] This class is new to DoJa-3.0 Profile
<code>ConnectionException</code>	This class throws an exception when an I/O processing error is detected in the <code>com.nttdocomo.io</code> package. Exception details are made available by the <code>getStatus()</code> method.
<code>PrintWriter</code>	Supports a text output stream for outputting the formatted object expression. This class offers functionality equivalent to a subset of the J2SE <code>java.io.PrintWriter</code> class. [DoJa-3.0] This class is new to DoJa-3.0 Profile.

Chapter 5 describes client-server programming using `HttpConnection` in detail. Also, for more details regarding OBEX external-connection programming, refer to Chapter 10.

Package `com.nttdocomo.net`

The package contains the following classes.

Name	Description
<code>URLEncoder</code>	This class contains a utility method for encoding a <code>string</code> in URL encode format (i.e., an encode format used with the <code>x-www.form-urlencoded</code> MIME type). Upon encoding, the <code>string</code> is converted to SJIS code.
<code>URLDecoder</code>	This class contains a utility method for decoding from URL encode format to a <code>string</code> . Input data is assumed to be based on SJIS code.

These classes have the same capabilities as the classes with the same names in package `java.net` of Java 2 Standard Edition.

Package `com.nttdocomo.util`

This package contains the following classes and interfaces:

Name	Description
<code>EventListener</code>	All event listener interfaces addressed by the i-appli API inherit this interface.
<code>JarFormatException</code>	Exception class indicating that a <code>Jar</code> file format error has occurred. This exception occurs when an attempt is made to handle data of an invalid format in the <code>JarInflater</code> class. [DoJa-3.0] This class is new to DoJa-3.0 Profile.
<code>JarInflater</code>	Provides functionality for expanding and extracting entries contained in <code>Jar</code> format file images. The <code>Jar</code> file image can use either a byte array or byte stream. [DoJa-3.0] This class is new to DoJa-3.0 Profile.
<code>MessageDigest</code>	Provides functionality for requesting a message digest (hash value) from any byte data. The MD5 and SHA-1 algorithms are supported in DoJa-3.0 Profile. [DoJa-3.0] This class is new to DoJa-3.0 Profile.

Name	Description
Phone	Representing a dialer, the <code>Phone</code> class provides a means for accessing the mobile phone's native voice-call functionality. When this voice-call functionality is called from an i-appli, the JAM will ask the user to confirm whether voice-call transmission can be carried out. It is not possible to omit this confirmation process. This class also has methods for retrieving the dialer's unique identification information (information unique to the hardware, such as the serial number). [DoJa-2.0] This class is newly added with DoJa-2.0 Profile. [DoJa-2.1] Methods for retrieving unique identification information were added in DoJa-2.1 Profile.
ScheduleDate	Class for expressing the schedule time. This class is used to set the schedule time to the native application from i-appli. With this class, schedules can be entered for native applications, such as the scheduler and alarm settings. [DoJa-3.0] This class is new to DoJa-3.0 Profile.
TimeKeeper	The i-appli API supports a timer function that notifies the i-appli when a set period of time has elapsed in the form of an event. Timer classes (<code>com.nttdocomo.util.Timer</code> and <code>com.nttdocomo.ui.ShortTimer</code>) implement this interface. The timer's resolution (smallest timer interval that can be handled) depends on the manufacturer. The timer resolution can be learned using the <code>getResolution()</code> method defined by this interface.
Timer	This is the timer class. This class supports one shot timer and interval timer. Object <code>TimerListener</code> can be registered for receiving timer events. Method <code>timerExpired()</code> of the listener object is called when the timer event occurs. The period for which the timer is valid spans from the launching of the i-appli to its termination, and timer operation will continue even if the screen is replaced. However, the timer will stop if i-appli execution is interrupted or, in the case of a stand-by application, goes into the sleep mode.
TimerListener	This interface is for implementing a listener for getting timer events notified by <code>com.nttdocomo.util.Timer</code>. Note that since <code>com.nttdocomo.ui.ShortTimer</code> uses method <code>processEvent()</code> of <code>Canvas</code> for its event processing, it does not use this interface.

Package com.nttdocomo.lang

This package contains the following exception class:

Name	Description
<code>IllegalStateException</code>	If calling of operations or methods is carried out in conditions for which this is not permitted, <code>IllegalStateException</code> will be thrown as an execution exception. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.
<code>UnsupportedOperationException</code>	<code>UnsupportedOperationException</code> is an execution exception that is thrown when an unsupported operation or method is executed.

Package com.nttdocomo.ui

This package contains the following interfaces:

Name	Description
<code>ComponentListener</code>	The listener interface for receiving component events. Classes that process component events must implement this interface. The listener object is registered in the <code>Panel</code> using the <code>Panel.setComponentListener()</code> method. When a component event takes place, the <code>componentAction()</code> method of the listener object is invoked, passing as arguments the type of event, the component in which the event took place, and event parameters.
<code>FocusManager</code>	The focus manager class is an object that manages the transfer of focus between components placed on the <code>Panel</code>. This is the interface implemented by the focus manager class. When no specific indications are carried out for the focus manager, the default focus-manager class will be used. In all previous profiles, no focus manager other than the default focus manager was defined.
<code>Interactable</code>	Interface implemented for components (e.g. <code>Button</code>, <code>ListBox</code>, and <code>TextBox</code>) that can be operated by the user. This interface defines a method for setting component focus and for setting whether components can be operated.
<code>KeyListener</code>	A listener interface for receiving individual key operations in the high-level API. Objects implemented by this interface are registered in the <code>Panel</code>. This interface defines 2 methods, corresponding to key press events and key release events: <code>keyPressed()</code> and <code>keyReleased()</code>.
<code>LayoutManager</code>	The layout manager is an object that manages the layout of components on the <code>Panel</code>. This is the interface implemented by the layout manager class. i-appli Runtime Environment includes definitions for the default layout manager to be used in situations where no specific indications are carried out for the layout manager, and also for the HTML layout manager used for HTML-type layouts. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.
<code>MediaData</code>	An implementation object of <code>MediaData</code> is obtained by using the <code>MediaManager.getData()</code> method. The actual object to be returned is implementation-specific. The <code>MediaData</code> interface was created for manufacturers to implement new proprietary media data.

Name	Description
MediaImage	MediaImage represents static or dynamic images. An implementation object of MediaImage is obtained by using the <code>MediaManager.getImage()</code> method. MediaImage conducts playback using the <code>VisualPresenter</code> object. Usable data will be of the GIF format (GIF87/87a/89a) or the JPEG format (i.e., JFIF or EXIF format with Baseline encoding) [Doja-2.0] GIF format is the only usable data in Doja-1.0 Profile.
MediaListener	This interface is implemented by the media listener in order to receive events from the media presenter that presents multimedia data. The <code>mediaAction()</code> method defined by this interface can receive such media presenter events as start playback, terminate, and stop.
MediaPresenter	Defines the interface that an object presenting media data must implement.. The presenter class presents media data according to its actual content. Supported media data types are terminal-dependent. If media data not supported by the terminal is set, a <code>UIException</code> is thrown.
MediaResource	Defines the interface that media data interfaces (<code>MediaData</code>, <code>MediaImage</code>, and <code>MediaSound</code>) need to inherit.. The <code>use()</code> method must be used before a <code>MediaResource</code> instance obtained by using <code>MediaManager</code> can actually be used, and the <code>unuse()</code> method must be called after use. The <code>dispose()</code> method should be called when the <code>MediaResource</code> instance is no longer needed, in order to discard that instance.
MediaSound	<code>MediaSound</code> represents sound. An implementation object of <code>MediaSound</code> is obtained by using the <code>MediaManager.getSound()</code> method. <code>MediaSound</code> is played using an <code>AudioPresenter</code> object. On an actual mobile phone, sound data of the MFi (Melody For i-mode) format can be used.
SoftKeyListener	This interface is used to listen for the two soft keys on the mobile phone being invoked in the high-level API. Implementation objects of this panel must be registered on the <code>Panel</code> in order to be used. This interface defines two methods, corresponding to a soft key pressed event and soft key released event:: <code>softKeyPressed()</code> and <code>softKeyReleased()</code>.

This package contains the following classes.

Name	Description
AnchorButton	The AnchorButton component represents HTML anchor-type buttons (i.e., underlined text which can operate as a button). It is possible to specify an image to be used as a heading in front of the text; furthermore, text can extend over multiple lines. In situations where processing is to be carried out in response to pressing of an anchor button, Listener registration of the object implementing ComponentListener is to be carried out using Panel's setComponentListener () method. Listener will then accept BUTTON_PRESSED events. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.
AudioPresenter	This class is used to play back sound media data (MediaSound). The AudioPresenter can be used both when using high level API and low level API.
Button	The Button component is represented as a labeled button. In situations where processing is to be carried out in response to pressing of a button, Listener registration of the object implementing ComponentListener is to be carried out using Panel's setComponentListener () method. Listener will then accept BUTTON_PRESSED events.
Canvas	A Canvas is a blank rectangular area of the screen, onto which the i-appli can draw or from which the i-appli can trap input events from the user. The developer will typically define a subclass inheriting from Canvas. The i-appli overrides the paint () method of the Canvas class to draw on the Canvas.
Component	A Component is a user interface object having a graphical representation that can be displayed on the screen and be used to interact with the user. Examples of Component subclasses are Button, ListBox, and TextBox. Component belongs to the high-level API, and is used in combination with the Panel.
Dialog	A Dialog is top-level window with a title, a text message, and at least one button. A Dialog can be displayed on top of either the Panel or a Canvas.
Display	Display is a class that manages screen real estate. All methods of Display are static, and no instances are ever created. The Display class changes screen content (Panel or Canvas) and delivers input events to the displayable object currently shown on the screen. This class receives low-level events and delivers them to the appropriate displayable object, which in turn decides how to process those events. Display also contains device information, such as the width/height of the display area, as well as a variety of constants for use with keys and event processing.
EncodedImage	Class which holds the results of encoding the Canvas or Image rendered content to a byte stream (JPEG format, etc.) with the ImageEncoder class. i-appli can use this class to extract a byte image from the rendered contents, or reconvert and extract to a media image. [DoJa-3.0] This class is new to DoJa-3.0 Profile.

Name	Description
Font	Shows the font used when displaying text in the <code>Graphics</code> context, <code>Component</code> or <code>Dialog</code>. Which fonts are supported depends on the manufacturer. [DoJa-3.0] Features for setting a font to <code>Component</code> and <code>Dialog</code> were added in DoJa-3.0 Profile.
Frame	<code>Frame</code> is an abstract class that provides a screen representation. The content that can be shown on the display of a mobile phone at any one time corresponds to a <code>Frame</code> instance. Some examples of <code>Frame</code> subclasses are <code>Canvas</code>, <code>Dialog</code>, and <code>Panel</code>. To change a screen, call the <code>Display.setCurrent()</code> method with the <code>Frame</code> object for the desired screen set as the parameter. (Note, however, that in the case of <code>Dialog</code> only, display is achieved using the <code>Dialog.show()</code> method.)
Graphics	This class carries out rendering for <code>Canvas</code> and <code>Image</code>, and it also encapsulates the corresponding operations. Note that <code>Image</code> rendering can only be carried out for <code>Image</code> instances where the corresponding image has been newly created using the method <code>Image.createImage()</code>. [DoJa-2.0] Functionality for the acquisition of <code>Graphics</code> instances from <code>Image</code> instances and for rendering are newly added with DoJa-2.0 Profile.
HTMLLayout	This layout manager is used to create HTML-type layouts. Although similar to the default layout manager in terms of layout creation in accordance with the sequence of addition to <code>Panel</code>, this manager can use its methods to make indications for HTML carriage return, paragraph separation, and alignment. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.
IApplication	The <code>IApplication</code> class is the i-appli superclass. An i-appli's main class (the first class called when an i-appli is launched) must extend <code>IApplication</code>. This class manages the life-cycle from start to end of an i-appli.
Image	The abstract class <code>Image</code> represents static images. A <code>MediaImage</code> instance is created from a static (GIF or JPEG) image, and an implementation instance of <code>Image</code> can be obtained by calling the <code>getImage()</code> method. The <code>Image.createImage()</code> method can be used to create new <code>Image</code> instances. [DoJa-2.0] The <code>createImage()</code> method is newly added with DoJa-2.0 Profile; furthermore, JPEG format is not supported in DoJa-1.0.
ImageButton	The <code>ImageButton</code> component realizes buttons which may be pasted onto static images. In situations where processing is to be carried out in response to pressing of an image button, Listener registration of the object implementing <code>ComponentListener</code> is to be carried out using <code>Panel.setComponentListener()</code> method. When the image button is then operated, listener will accept <code>BUTTON_PRESSED</code> events. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.

Name	Description
ImageEncoder	Provides functionality for encoding the rendered content of the Canvas or Image as a JPEG etc. format byte image. Results of encoding are returned as an EncodedImage instance. The common encoding format that can be used in all phone models is JPEG. [DoJa-3.0] This class is new to DoJa-3.0 Profile.
ImageLabel	An ImageLabel component is used to display a static image on the Panel. The Image instance can be set to ImageLabel, and placed on the Panel.
Label	The Label component is used for displaying text on the Panel. A Label represents a single line of read-only text. The text can be changed by the application program, but a user cannot edit it directly.
ListBox	The ListBox component presents the user with a scrolling list of text items. The developer can choose the appearance and capabilities of the ListBox by making the appropriate specifications when creating the instance.
MApplication	MApplication constitutes a template class for stand-by applications. This class inherits the class IApplication, and the main class for the stand-by application (of the class initially called upon startup of the stand-by application) must be a sub-class of this class. The MApplication class manages the life-cycle of stand-by applications through launch, termination, and execution conditions. [DoJa-2.0] This class is newly added with DoJa-2.0 Profile.
MediaManager	A class for managing the MediaResource. The MediaManager is the factory class for media data (MediaData, MediaImage, and MediaSound). Media data can be created by specifying a URL location string, stream or byte array. [DoJa-3.0] Creation of media data by specifying a stream or byte array was added in DoJa-3.0 Profile.
Panel	Panel is a container class for displaying components on the screen in the high-level API. A Panel displays space can attach any component. Only one Panel can be displayed on the screen at a time.
PhoneSystem	This class is used to access the native resource of platform. The platform's native resources include icon information (mail icon, etc.) shown on the display. [DoJa-3.0] Icon information referencing was added in the DoJa-2.0 profile.
ShortTimer	This is a relatively low-overhead timer meant for use in the low-level API. This class supports one-shot timer and interval timer. ShortTimer is obtained by using the getShortTimer() static method of this class. Canvas is passed as the argument to the getShortTimer() method, and the timer event passes this Canvas to the processEvent() method. ShortTimer only operates when the corresponding Canvas is being shown on the display. If a frame change occurs while ShortTimer is operating, the timer automatically stops.
TextBox	The TextBox component is used to display and input text.. The TextBox is the only component which allows the user to enter any character string.

Name	Description
<code>Ticker</code>	The <code>Ticker</code> component is used to display streaming text that automatically scrolls to the left. The component displays text in a ticker tape style (equivalent to the MARQUEE tag in i-Mode HTML).
<code>UIException</code>	<code>UIException</code> is a runtime exception class that is thrown when an error is detected while a UI package, a native function using the UI package, or similar software is running.
<code>VisualPresenter</code>	The <code>VisualPresenter</code> component is a presenter class used for playing the visible media (<code>MediaImage</code>) data in the high-level API.

Package `com.nttdocomo.device`

This package contains the following classes for controlling the mobile phone's peripheral devices.

Name	Description
<code>Camera</code>	Calls the mobile phone's camera feature, and provides a means for accessing functionality for shooting still and moving images. The actual shooting is performed by the user operating the shutter with respect to the camera feature activated by the i-appli. Support for the shooting of moving images depends on the manufacturer.
<code>DeviceException</code>	<code>DeviceException</code> is the runtime exception class thrown when an error is detected, mainly during the operation of the device package and native functionality that use the device package.
<code>IrRemoteControl</code>	Provides infrared remote control functionality. This class acts as a transmitter in order to send the remote control signal indicated in the <code>IrRemoteControlFrame</code> class from the infrared port.
<code>IrRemoteControlFrame</code>	Indicates the data frame indicating the external device control details sent from the infrared remote control feature. In general the manufacturer of that external device decides independently what control is performed when a given kind of data-frame is received. It is necessary for the programmer to design the application so that the data frames transmitted comply with the specifications of the targeted external device.

Package com.nttdocomo.system

This package contains the following classes and interfaces for utilizing the phonebook feature, bookmarks, scheduler and other native functionality of the mobile phone from i-appli. This package contains many classes and methods the use of which are permitted only in the trusted i-appli. These features will not be explained here.

Name	Description
ApplicationStore	Provides functions that access the mobile phone's native i-appli management functions. By using this class, the entry ID, which is required when starting another i-appli that has been downloaded to the mobile phone (launcher mode), can be retrieved.
Bookmark	Provides functions that access the mobile phone's native bookmark management functions. Using this class, bookmark entries can be newly added to the mobile phone.
ImageStore	Provides functions that access the mobile phone's native image data management functions. Using this class, images can be read from and newly added to the mobile phone's image data storing area.
InterruptedException	Exception indicating that the user's operation of a native application function called by the i-appli was interrupted by an outside event, such as an incoming voice call.
PhoneBook	Provides functions that access the mobile phone's native phonebook management functions. By using this class, phonebook entries can be newly added to the mobile phone.
PhoneBookConstants	Interface which encompasses all constants related to phonebook management functions. This interface is implemented in the PhoneBook and PhoneBookParam classes. The developer need not create this interface's implementation class.
PhoneBookGroup	Provides functions that access the phonebook group management functions of the mobile phone's native phonebook group management functionality. By using this class, new phonebook groups can be newly added to the mobile phone. New phonebook entries can be added to the phonebook groups created with this class by using the PhoneBook class.
PhoneBookParam	Indicates phonebook entries for use as new additions. Phonebook entry data, such as name and phone-number, are assigned to this object and it is added to the mobile phone with the PhoneBook class.
Schedule	Provides functions that access the mobile phone's native scheduler. By using this class, new schedules can be added to the mobile phone.
StoreException	Exception indicating an access error, such as insufficient empty area, occurred when attempting to access a data area managed by a native mobile phone application.

Note:

- Within the com.nttdocomo.system package classes, there are many functions which call a native mobile phone function and newly add user data. The mobile phone's user can disable these functions through the mobile phone settings. Also, when calling these functions, the i-appli runtime environment requires user confirmation with regards to the registration of this data. If the user's permission is not received through this mechanism, that data cannot be added.

IAApplication and MApplication

The `com.nttdocomo.ui.IApplication` class is the superclass of all i-applis. Typically, an i-appli's main class (the class specified by the ADF's `AppClass` key, and is the first class called when the i-appli is launched) must inherit from this class. This is similar to the case of Java applets running on a PC browser, which must inherit from the `java.applet.Applet` class.

The `IApplication` class defines the key methods relating to an i-appli's lifecycle, from launch to termination:

- The `IApplication.start()` method begins the execution of an i-appli. The developer overrides this method, and implements its processing classes. At this time, it is possible to call the `IApplication.getArgs()` method to obtain the i-appli's startup parameter (value specified by the ADF's `AppParam` key).
- When first launching after an i-appli download, or when instructing an i-appli launch with application linking functionality, parameters can be passed to i-appli from HTML or the mail body.
- Call the `IApplication.terminate()` method to terminate an i-appli's execution.
- If a high-priority external event, such as an incoming phone call, occurs during i-appli execution, i-appli is interrupted and handling is performed for the external event. When handling for the external event completes, i-appli automatically restarts. When i-appli restarts, the `IApplication.resume()` method is called. The developer can implement any handling needed when resuming i-appli execution by overriding this method. Further, by using the `IApplication.getSuspendInfo()` method (added in the DoJa-3.0 profile), the reason for previous i-appli interruptions, and information on the events that fired during the interruption can be retrieved (depending on the manufacturer, it may be possible to retrieve information on events that fire during just that one interruption occurrence).
- By using the `IApplication.launch()` method, another native application or i-appli can be launched from i-appli. If this operation is successful, the original i-appli will end execution.

Furthermore, the `com.nttdocomo.ui.MApplication` is inherited and extended from `IApplication` — the class which provides support for the complicated life-cycle unique to stand-by applications. The main class for stand-by applications must inherit from the `MApplication` class.

In addition to the functionality of `IApplication`, the class `MApplication` also includes the following functions.

Stand-by Application Mode Control:

Stand-by applications can adopt a total of three modes — namely, the same Active mode as adopted by other i-applis during execution in addition to the modes listed below.

- Inactive mode: Although an i-appli is operating, hooking occurs as a result of the mobile phone's native stand-by function being used by key events or soft-key events, and no notification is provided to the i-appli.
- Sleep mode: Screen display is terminated without affecting i-appli operation in order to conserve power.

The `MApplication` class manages transition between these conditions.

System Event Handling:

In the case of stand-by applications, events from the outside are notified to resident i-applis as system events. The developer can override system event handlers with the `MApplication` class so that, regardless of which of the above-mentioned three conditions is currently adopted, it will be possible to receive and process said events.

For more details regarding stand-by applications, refer to Chapter 9.

Notes:

- The application does not have a direct interface between `IApplication` and its subclasses. Instances of these are created by the i-appli Runtime Environment.
- If your i-appli explicitly defines a main-class constructor, be sure to make the access specifier public. With the i-appli Runtime Environment, furthermore, it is not possible to use constructors with arguments when creating a main-class instance. If you explicitly define a main-class constructor, be sure to confirm that it does not have any arguments.
- CLDC defines the following methods: `java.lang.Runtime.exit()` and `java.lang.System.exit()`. In the i-appli Runtime Environment, however, these methods cannot be used to terminate an i-appli's execution (if these are called, a `SecurityException` is thrown). Call the `IApplication.terminate()` method to terminate an i-appli's execution.

Text Conversion

i-appli will process text using the Japanese language character set and the ASCII character set. The `java.io` package provides classes that allow you to convert between Unicode character streams and byte streams of non-Unicode text. You use the `InputStreamReader` to convert byte streams to character streams, and the `OutputStreamWriter` to convert character streams to byte streams. See Chapter 6 for more details on text conversion.

ScratchPad

The storage model that mobile phone i-applis use is called the ScratchPad. The `open()` method of the `Connector` class is used to connect to the Scratchpad. Passing the URL for the ScratchPad to this method opens a connection, and returns an implementation instance of the `javax.microedition.io.StreamConnection` interface defined by CLDC. The `StreamConnection` interface is used to read and write to the ScratchPad area. See Chapter 7 for more details on using the ScratchPad.

Chapter 3

Design Considerations

This chapter provides guidance on designing i-applis that efficiently use available system resources, are well adapted to the wireless communications environment, and are easy to use.

3.1 Characteristics of i-appli Mobile Phone

In the interests of minimizing the physical size and maximizing the battery life of the mobile phone, the mobile phone's display screen, application executable memory, and application storage memory are very small by comparison with personal computers. Also, the processor runs at a slower speed. The Liquid Crystal Display (LCD) screen consumes very little electricity, but has a slower refresh rate than other types of displays. Text entry must be done using the mobile phone keypad, and since more than one character is assigned to each key, entering text will be a tedious procedure for the user. Text input should therefore be kept to a minimum.

When designing services which make use of i-appli, it will be important to take these platform-specific characteristics into consideration during i-appli design. The following sections discuss these issues from a design standpoint, and offer some guidelines for designing i-applis that are easy to use.

3.2 Memory Issues

Figure 6 shows how the JAM allocates storage for 3 i-applis (storage for the JAR file and ScratchPad).

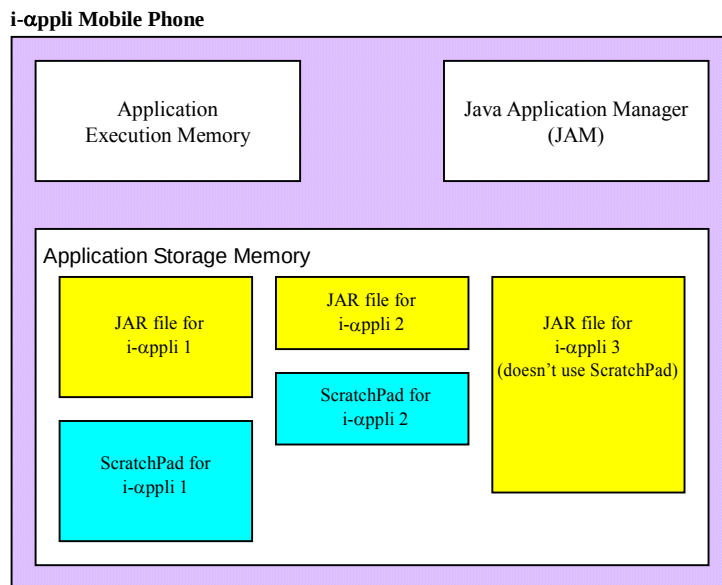


Figure 6: i-appli Mobile Phone Memory

As a content developer, you are concerned with the following two memory constraints:

- Limited application execution memory in which to execute your i-appli and store objects created at runtime.
- Limited application storage memory in which to store your i-appli (JAR file) and the ScratchPad associated with i-appli.

The absolute limit on application runtime size is the amount of application execution memory in the mobile phone in which the i-appli is to execute.

The limit on application storage size is 30 Kbytes per one JAR file (hereafter, 1 Kbyte = 1,024 bytes), and 200 Kbytes for the ScratchPad. Note that i-applis exceeding these limits will not download successfully to the mobile phone.

Note:

- Some manufacturers support ScratchPad sizes of over 200 Kbytes. Also, the area management method employed by some manufacturers limits the size of the ScratchPad of an i-appli to be installed by the sum of the ScratchPads that are already installed. In either case, if the mobile phone cannot secure the ScratchPad capacity declared by the i-appli's ADF `SPsize` key, the i-appli can't be downloaded to the mobile phone.

[DoJa-2.0] [DoJa-3.0]

DoJa-1.0 Profile defined the maximum sizes for JAR files and for the ScratchPad as 10 Kbytes and 5 Kbytes respectively. In contrast to this, in DoJa-2.0 Profile, the JAR file maximum is extended to 30KB, and the ScratchPad maximum to 100KB. In DoJa-3.0 Profile, the ScratchPad has been further extended to 200KB. In DoJa-3.0 Profile, a maximum of 16 ScratchPads assigned to a single i-appli can be internally segment-managed. On how to access a segment-managed ScratchPad, see chapter 7.1 and to see how to set the ADF to segment-manage the ScratchPad, see 13.5.1.

3.2.1 Minimizing Memory Usage

The discussion comprises two broad categories: application execution and application storage memory.

Application Execution Memory

Typically, the largest component in application execution memory is the executable program itself (the classes comprising the i- α pli Runtime Environment and the i- α pli). If the application, with all of the objects it needs, cannot fit in the available execution memory, then you have no choice but to reduce the size of the executable program, the size of the set of generated objects, or both.

For information on reducing the size of the executable program, see Section 3.2.2.

To reduce the size of the set of generated objects, consider the following:

- Calling the garbage collector often. Particularly with the i- α pli Runtime Environment and its small memory capacity, the fragmentation which occurs in the application execution memory often leads to cases where the remaining memory cannot be accessed as a single item. (Exceptions such as `OutOfMemoryError`, `UIException`, and `ConnectionException` will occur, and this will manifest itself in unstable i- α pli operation.) In such a case, the garbage collector should be called as soon as possible to resolve any problems.
- Keeping as few objects in memory as possible. Also, minimizing the number of objects and size you generate.
- Consider breaking up the application (for example, in designing a game with multiple levels, make multiple i- α pli, dividing the application program up by game levels.
- Using static when applicable
- Minimizing stack depth by avoiding excessive nested method calls or recursion.
- Minimizing number of parameters to methods
- Minimize the number of classes and the size of individual classes
- Avoiding the proliferation of threads
- Creating classes with pre-set size. Dynamically growing classes such as `HashTable`, `ByteArrayInputStream`, `ByteArrayOutputStream`, or `Vector` may lead to memory fragmentation.
- Dispose of any media resources and image that are no longer required (`dispose()`).

If the objects you are generating are as small as you can make them, and you are using application execution memory efficiently but your program still runs out of memory, consider the following:

- Moving data stored in variables to the ScratchPad.
- Off-load processes which consume a lot of memory from the client to the server.
- If downloading data from a server, avoid doing this in one batch, but rather carry out partial downloads of the smallest possible size.

Application Storage Memory

Your application consumes two blocks of application storage memory to save the i- α pli:

- The block allocated to the JAR file.
- The block allocated to the ScratchPad.

JAR File

Make sure the JAR file contains only essential files, and that those files are as compact as possible. If your i-appli contains different modules for each mobile phone, for instance if using i-appli optional APIs and i-appli extension APIs, consider creating a separate i-appli (a separate JAR file) for each device (see Chapter 14 for details on how to control device-specific downloads). That way, the JAR file downloaded to the mobile phone will contain no extraneous entries.

Also, although you can include resources like images and sound in the JAR file (see Chapter 7), this has a large impact on its size. Consider using the mobile phone's communication capabilities to obtain infrequently used resources at i-appli runtime.

Also note that the i-Mode service employs a packet billing system for charging communication fees. Reducing the size of the JAR file will make it cheaper for the user to download the i-appli.

ScratchPad

Do not specify a larger ScratchPad than necessary. And regardless of how much space you allocate to the ScratchPad, use that space as efficiently as possible. You specify your ScratchPad size in the i-appli's ADF (see Section 13.5.1 for details). Also note that the available size of the i-appli's ScratchPad is subject to the limitations discussed in Section 3.2.

The kind of physical device (hardware) used in the ScratchPad depends on the manufacturer. Note that depending on the model of phone, allocating a larger than necessary ScratchPad could cause a decrease in access speed and impact i-appli performance.

3.2.2 Class File Size

Compiled Java class files sometimes contain debug information, which increases the size of the file. Reduce class file size by keeping the number of classes defined down to a minimum, and by compiling the code with the `-g:none` option to remove debug information.

Note that if you are building i-applis using the i-appli Runtime Environment build function as provided by NTT DoCoMo, the `-g:none` option will be set during compilation. For more details regarding the i-appli Runtime Environment, refer to Chapter 13.

3.3 Usability and GUI Design

This section presents suggestions for creating i-applis that are easy and intuitive to use. In general, it is recommended that on a platform like the mobile phone, complex i-applis requiring a lot of keystrokes be avoided in favor of i-applis with fewer keystrokes. A few points to remember are:

- Maintain a consistent look and feel throughout the i-appli
- The physical screen area is small, so keep the number of UI components shown on a single screen to a minimum. In particular, if a number of UI components supporting user operation (i.e., buttons, etc.) are displayed on a single screen, the requirement for focus-shift operations can result in users considering to be difficult to use.
- UI layout management will be manufacturer dependent, and in most cases the default layout management will be simple flow layout. UI layout is kept simple, and the i-appli execution screen's appearance is kept uniform, without drastic differences between various manufacturers.
- Keep the scrolling to a minimum. Note that while vertical scrolling is supported by `Panel`, it will be necessary to control screen scrolling from the application program when using `Canvas`.
- Softkeys are a useful way to simplify user operations. Consider using softkeys instead of Buttons on screens that generate complex user operations. For instance, you could use a softkey to select a component on a screen upon which a large number of input component objects are placed, regardless of which component has focus.
- Text entry is complex and difficult on a mobile phone, so minimize the number of characters the user must enter. Where possible, use an input method that presents choices and lets the user selected from among them.
- Design your screen composition so that commonly performed tasks do not require the user to navigate through many menu levels, or to enter many keystrokes.

3.3.1 Designing for the Target Audience

Everyone uses a phone, but not everyone uses computers, so the concepts of scrolling and moving between GUI components are not intuitive to every user. Because there will be a broad range of users, user interface operations should be as intuitive as possible. When designing i-appli user interfaces, it is necessary to ascertain the target users' level.

3.3.2 Design Policy for i-appli Operation Methods

Divide user tasks into categories, based on how commonly they will be performed and how valuable they are to the user. In general, you should make it very easy for the user to perform common tasks. For example, checking a bank balance will be performed more often than transferring money between accounts, so ideally the account balance function would have fewer keystrokes compared to the transfer function.

On the other hand, a task that requires several keystrokes – such as entering a Personal Identification Number (PIN) – will certainly be accepted by the user if it performs an important function (in this case, protecting the user's bank account or other assets from unauthorized access). And even a certain amount of user input could be acceptable if a little tedium when first using an i-appli after it is installed on the mobile phone will simplify later activities.

Activities should be consistent. Selecting an item on one screen should be the same as selecting an item on another. Softkeys should be used with a consistent concept throughout the i-appli in order to avoid user errors and frustration. The most important and often-used tasks should require the fewest keystrokes.

3.3.3 Scrolling

You can theoretically design a virtual screen that is larger than the physical screen. However, this will require a large amount of scrolling to view the entire screen, or to shift focus between UI components - and since scrolling is not always the most responsive of screen operations, screens with a lot of scrolling could frustrate the user. Your goal should be to minimize scrolling.

[DoJa-2.0]

With DoJa-1.0 Profile, it was left up to individual manufacturers to decide whether or not to support Panel scrolling. With DoJa-2.0 Profile or later, however, a Panel for which the default layout manager or the HTML layout manager has been set will support scrolling in the vertical direction.

3.3.4 Keys and Softkeys

i-appli can make use of the following standard keys.

- Direction keys and the selection (confirmation) key
- Number keys, 0 through 9, in addition to the # and * keys
- Stand-by application switching key

Certain manufacturers may support other special keys to provide functionality for the i-appli option category.

Furthermore, i-appli capable mobile phones also feature two more soft keys other than the above. The appearance and location of the softkeys depend on the manufacturer. As stated earlier, using softkeys makes a user interface much more pleasant to use. While you should consider using softkeys in your i-appli design, you should avoid relying on a specific appearance or location for them.

[DoJa-2.0]

The stand-by application switching key has been newly implemented in DoJa-2.0 Profile in order to support stand-by application functionality. With the exception of cases where the stand-application is in either the sleep or inactive condition, it will be possible for the i-appli to make direct use of this key (and to obtain key events). If in the sleep condition or inactive condition, events from the stand-by application switching key will be used by the system.

For more details regarding stand-by applications, refer to Chapter 9.

3.3.5 Data Entry

The mobile phone's keypad is the user's only means of entering data. Numeric data is therefore easier to enter compared to alphanumeric input. This is because in the case of text data, it takes multiple keystrokes to enter a single character. Text entry should therefore be kept to a minimum.

3.3.6 Password Entry

Perform password entry in either of the following ways:

- Allow hidden passwords. In the `TextBox` constructor, pass `TextBox.DISPLAY_PASSWORD` to the display mode parameter. All input characters are shown as the '*' character.
- Allow passwords to be shown. In the `TextBox` constructor, pass `TextBox.DISPLAY_ANY` to the display mode parameter. All input characters are displayed on the screen.

If using the first of these methods, it is standard only for numbers to be used. In contrast to the case with PCs, if characters other than these are included in passwords, it will result in frustrating operations for the user. If using only numbers, the `TextBox.setInputMode()` method can be used to setup number-input mode as the initial input mode, thus preventing incorrect operation by the user.

When the use of non-characters (i.e., text characters) is to be permitted for passwords, the second method should be used.

3.3.7 Use of Threads

Threads are very useful for performing long, blocking operations such as network I/O or UI interaction. However, in any multi-threaded system, there is overhead associated with the context switching between active threads. Since the phone devices are limited in processing power, keep the number of threads to a minimum.

Furthermore, even if multi-thread programming has not been carried out for the application program side, the system may generate threads for event processing or the like. Regardless of the model on which they will operate, thread-safe program design must be implemented when creating i-applis.

Notes:

- In the i-appli runtime environment, dialogs or pop-ups will be displayed under the following circumstances:
 - The application itself uses the `Dialog` class to display a dialog.
 - In HTTP(S) communications, the Web server requests authentication (BASIC authentication)
 - In HTTP(S) communications, when displaying a warning to the user because there is a problem with the server's certificate.
 - When using the infrared port for OBEX external-connection or infrared remote control functionality.
 - When confirming the launch of another application via application linking functionality with the user.

These dialogs and pop-ups can only be displayed exclusively, and cannot be displayed simultaneously from multiple threads. If an attempt is made to display these from multiple threads simultaneously, the resulting behavior varies depending on the phone's manufacturer. It is necessary for developers to design application programs so that the above conditions do not occur simultaneously in multiple threads.

3.4 Security

When designing i-applis with sensitive information passed between client and server, consider using the HTTPS protocol. Note that when this mechanism is used on the client side, the burden on the CPU increases. This is because at the beginning of every SSL session, on the client side (in this case, the i-appli) in addition to having to create master keys, data encryption and decryption based on these keys must be performed whenever data is processed and sent to or received from the server.

See Section 5.3 for information about communications using the HTTPS protocol.

Note that when an i-appli communicates with a server, the server must have the same protocol, server address, and port number as the location from which the i-appli was downloaded. Consequently, i-applis which are to use the HTTPS protocol must be downloaded using this protocol.

3.5 Operating Over a Wireless Network

Be aware that the i-applis you develop will be used in an environment in which interruptions in the communications connection are commonplace (User's frequently leave and re-enter areas with mobile phone signals available.). The i-Mode service employs a packet billing system, so i-applis should transmit only the necessary amount of data when communicating.

In designing client/server i-appli, it is necessary to strike a balance between sending/receiving a number of small data packet transactions (minimizing data loss due to interruptions) or sending/receiving one large data packet (minimizing communications overhead).

See Chapter 5 for a full discussion of the issues related to establishing, testing, and terminating connections and managing transactions and sessions. See also the "Process Suspension and Resumption" section in this chapter.

3.6 Process Suspension and Resumption

The JAM automatically suspends a running i-appli when the phone receives an incoming voice call during i-appli runtime. The JAM then sends a Resume message to the i-appli and restarts program execution when the user terminates the voice call. In this way, in cases where it is necessary for a different function with a higher priority to operate on a mobile phone, the suspension and resumption of i-applis may be carried out.

In addition to the receipt of an incoming voice call, certain other conditions exist for which i-applis will be suspended. The following presents cases where i-applis are suspended and resumed.

- Incoming voice call during i-appli runtime (although if the i-appli is performing data communication at this time, certain i-mode incoming settings on the mobile phone will result in the application not being terminated.), or alternatively, receipt of mail or messages while the stand-by application is operating in inactive or sleep mode.
Note that when the stand-by application is operating in active mode, response will be the same as for execution of normal i-applis — in other words, the i-appli will continue to execute regardless of whether or not mail or messages are received.
- Another application launches during i-appli runtime. This includes both situations where a native application such as a time alarm is launched automatically during i-appli runtime and situations where the application link function is used to launch a native application from another i-appli. Note that in cases where the launched application is of the type which carries out packet communication (i.e., a browser or mailer i-appli), the calling i-appli will be shut-down rather than suspended.
- When a native function which entails user operation is called from i-appli. This occurs mostly in application linking functionality, and includes many features which require user confirmation.

- The user performs forced shut-down of an i-appli, and upon display of the confirmation dialog, the forced shut-down is canceled.
- The mobile phone includes other mechanisms for the suspension and resumption of i-applis (i.e., suspension / resume keys, etc.), and these are operated.

[DoJa-2.0]

Suspension and resume conditions for i-applis were clarified in DoJa-2.0 Profile. Please be aware that operations other than those described above may take place with DoJa-1.0 Profile.

[DoJa-3.0]

Interruption/resumption of i-appli has been added to basic functionality in DoJa-3.0 Profile.

Interruption/resumption of i-appli is not supported in some cellular-phones which support profiles older than DoJa-2.0 Profile. In these cellular-phones, when a condition arises requiring i-appli to be interrupted, it will be terminated.

When i-appli execution resumes, the i-appli main class (class inheriting `com.nttdocomo.ui.IApplication`) `resume()` method is called. The developer can implement i-appli resume processing by overriding the `resume()` method.

In some i-applis — for example, a standalone calculator application — you, as the developer, need not take any special steps to handle suspend/resume. This is because nothing inside will have changed since the i-appli was suspended and the progress of the application is entirely driven by user-generated events — namely, keystrokes. In particular types of i-applis, however, you must take special steps upon resuming execution. In general, there are a number of specific situations that require you to take some action upon resuming i-appli execution:

- If yours is a client-server i-appli, the communication result should be checked after the i-appli is resumed, and communication should be retried whenever so required. This action is necessary since a communication failure may occur upon the suspension or resume of the i-appli's communication processing.
- When an i-appli is suspended and resumed, the timers (i.e., `Timer` or `ShortTimer`) which had been operating before suspension will be stopped. It will be necessary for this type of timer to be explicitly restarted upon the resume of an i-appli.

Note that with DoJa-1.0 Profile, if the execution of an i-appli is suspended while a timer is being used, it is necessary after i-appli resume to discard the timer instance and to create it anew.

- If `AudioPresenter` was carrying out sound playback upon the suspension of an i-appli, `AudioPresenter` will be stopped, and even after resume of the i-appli, sound playback will not be automatically resumed. In this case, An explicit play command (i.e., `play()`) can be used from the i-appli to resume playback of the sound data from the start. Note that if `VisualPresenter` is carrying out playback of a moving image upon the suspension of an i-appli, this playback will be temporarily halted; however, this will automatically resume from the current frame when the i-appli is resumed.
- With DoJa-2.0 Profile, when `Canvas` is redisplayed as a result of the suspension or resume of an i-appli, it is guaranteed that the `paint()` method for the corresponding `Canvas` will be called once from the i-appli Runtime Environment. However, in cases where calling of this `paint()` method does not result in complete restoration of the entire screen (i.e., where the application program carries out only partial rendering in accordance with events), the degree to which the system can restore the screen will depend on the model in question.

In DoJa-3.0 Profile, when i-appli resumes, if the system can completely restore the rendered `Canvas` contents, the `paint()` method will not be called.

- In certain cases where an application is suspended and resumed, it will be desirable for a suitable prompt message to be displayed upon resume and for this resume to be temporarily halted. For example, in the case of action games and other applications which require immediate user interaction, the application's ease-of-use upon resuming may be improved by including a short period of time for the user to make the necessary preparations for resume.

[DoJa-2.0]

The effect of suspension and resume on i-applis has been clearly defined for DoJa-2.0 Profile. Please be aware that operations other than those described above may take place with DoJa-1.0.

[DoJa-3.0]

In DoJa-3.0 Profile, the `IApplication.getSuspendInfo()` method has been added in order to retrieve information on the reason for interruption and events which fired during interruption when i-appli interruption/resumption occurs.

Notes:

- Whether or not an i-appli performing data communication will be suspended upon the receipt of an incoming voice call will depend on certain i-mode incoming settings. If these settings dictate that incoming voice calls are not to be received during i-mode communication, then i-appli runtime will not be suspended even if an incoming voice-call is received while it is communicating data.
- In situations where unprocessed events are queued in the event queue upon suspension of an i-appli, all of these events will be discarded when the i-appli is resumed.

3.7 Accessing Hardware

The i-appli Runtime Environment provides a `PhoneSystem` class as a means for accessing the mobile phone's hardware (i.e., its platform resources). Using this class, it is possible to control items such as the display's back-light or the vibrator. Note that frequent access of platform resource an i-appli can consume a large amount of a mobile phone's battery power. Furthermore, even if the i-appli should change these settings, they will automatically be returned to their initial settings when the i-appli terminates.

[DoJa-3.0]

In DoJa-2.0 Profile or earlier, platform resource access functionality was contained almost totally in the i-appli Optional API, and was only partially implemented depending on the phone's manufacturer. In DoJa-3.0 Profile, most of this was added to the i-appli standard API.

3.8 Error Processing

If the CLDC API or the i-Mode Java extension API detects an error during processing, they throw errors and exceptions just like ordinary Java APIs. You will need to catch exceptions that the application program generates, and take appropriate action as necessary (for instance, submit an exception message to the user).

If runtime exceptions (`java.lang.RuntimeException` and its subclasses) and errors (`java.lang.Error` and its subclasses) are not caught by the application program, the i-appli will hang.

3.9 Stand-By Applications

DoJa-2.0 Profile or later provides support for stand-by applications as a means of implementing convenient resident-type i-applis such as clocks and the like.

Stand-by applications are resident-type i-applis which display a stand-by screen, possibly containing a stand-by image, while they are executing. Naturally, if the application program changes the image content, this will be immediately implemented in the stand-by screen.

In DoJa-1.0 Profile too, it was possible to show the user that an i-appli was a resident on the stand-by screen by constantly preserving the i-appli's executing state. However, this implementation method was characterized by the following problems.

- If it was desired to make a voice call or to browse the web, the user was required to explicitly shut down the i-appli. Furthermore, the user also had to explicitly restart the i-appli when these operations had been completed.
- It was not possible to receive mail while this type of i-appli was executing. For this reason, it was common for users to be unaware that mail had been received.
- Battery power continued to be consumed as the i-appli executed, and it was not possible to continue execution for extended periods of time.

Eliminating these types of problem, stand-by applications realize a means for providing users with convenient resident-type applications. If you are developing an i-appli for which constant, continuous operation is a pre-condition, consider the possibility of implementing this i-appli as a stand-by application.

Note that more details regarding stand-by applications are available in Chapter 9.

3.10 Identification of Cellular-Phone Models

The following two methods have been provided to enable an i-appli to identify the type of model on which it is executing.

- `microedition.platform`, a system property for i-appli Runtime Environment
An application program can use the method `java.lang.System.getProperty()` to reference system properties (i.e., the property set for acquisition of system information). Although the `microedition.platform` property is defined as a system property for indicating the host property in CLDC, the mobile phone's model name is stored as a property value with i-appli Runtime Environment. Accordingly, i-applis can check this number to identify the model on which it is currently executing.
The model name stored in this property will be identical to the name included in the user agent.
- User-Agent request data as transmitted to the server during HTTP communication
Similar to situations where browsers are being used for web access, a user agent (i.e., the User-Agent header) will be included in the request header when HTTP communication is being carried out from an i-appli. As a result, i-applis and those on the communicating server can check the content of the User-Agent header to identify the model of the currently-interacting mobile phone.

[DoJa-2.0]

With DoJa-1.0 Profile, the type of value to be set for `microedition.platform` is left to the discretion of individual manufacturers, and therefore, it will not be possible in all cases to obtain model names.

Chapter 4

Designing the User Interface

User Interface (UI) functionality has the following basic requirements:

- Creation of high-level User Interface components such as buttons and labels.
- Event processing.
- Layout processing.
- Image file display and sound file playback capabilities.

The i- α pli Runtime Environment User Interface API is made up of both low-level and high-level APIs. The following sections describe the basic areas of functionality and show code examples.

The high-level API provides application developers with UI components which, when assembled together, create the i- α pli User Interface. Applications using the high-level API have less concern for hardware characteristics, which allows them to easily run on different manufacturers' mobile phones. The developer has a limited degree of freedom, but on the other hand, can write high-function i- α plis with fewer lines of code. Items that are provided as user interface parts for the high-level API are referred to as "components." The developer assembles these components together to create an i- α pli. Screens using the high-level API generally have about the same level of expressiveness as i-Mode HTML.

The low-level API allows developers to create i- α plis whilst taking into consideration such hardware characteristics as the size of the mobile phone's screen or the keypad properties. The low-level API can be used to create i- α plis that make the most of the features of each mobile phone. The developer has a high degree of freedom, but must manage more detail, such as graphics rendering, screen size, and logical coordinates.

A single screen is displayed as a `Panel` or `Canvas` instance. `Panels` are used by high-level API screens (screens made up of components), and `Canvases` are used by low-level API screens (screens that are drawn "by hand"). A display can present only one `Panel` or `Canvas` instance; furthermore, it is not possible to use components in screens created using a `Canvas`. In screens created using a `Panel`, however, components for handling images can be used to incorporate low-level rendering.

A single i- α pli can contain a number of different `Panel` or `Canvas` instances, and by using the `Display.setCurrent()` method, an application program can switch the `Panel` or `Canvas` to be displayed.

Events such as a button press or a timer going off are handled differently by the high-level and low-level API. In the case of low-level API, all events can be received by the i- α pli, including key press/release events. With high-level API, however, depending on the component, some low-level events are handled internally, or converted into higher-level events and sent to the i- α pli.

In the case of low-level API, the low-level events which are generated as a result of actions such as key pressing or releasing, etc. are reported to the `Canvas` object. With low-level API, developers create and use sub-classes derived from the `Canvas` class, and by over-writing the `processEvent()` method from the `Canvas` class at this time, it is possible to incorporate low-level events. This method will be called when events occur, and therefore, processing for the event in question can be carried out internally. The type for the occurring event and parameters specific to that event are passed to the `processEvent()` method.

In the case of high-level API, high-level events resulting from button pressing and the like are reported to the `Panel` object. When processing for this type of high-level event is to be carried out within the i-appli, it will be necessary to register a `Listener` object in the `Panel` object. When events occur, the `Panel` object will call the `Listener` object registered within it, and it will report the events' occurrence. Different `Listener` interfaces for high-level events have been made available for different types of event-occurrence source. Furthermore, the information reported to the `Listener` event upon the occurrence of an event will vary with respect to the type of `Listener`.

Refer to Section 4.3 for more details on event listeners.

The figure below illustrates the classes making up the User Interface.

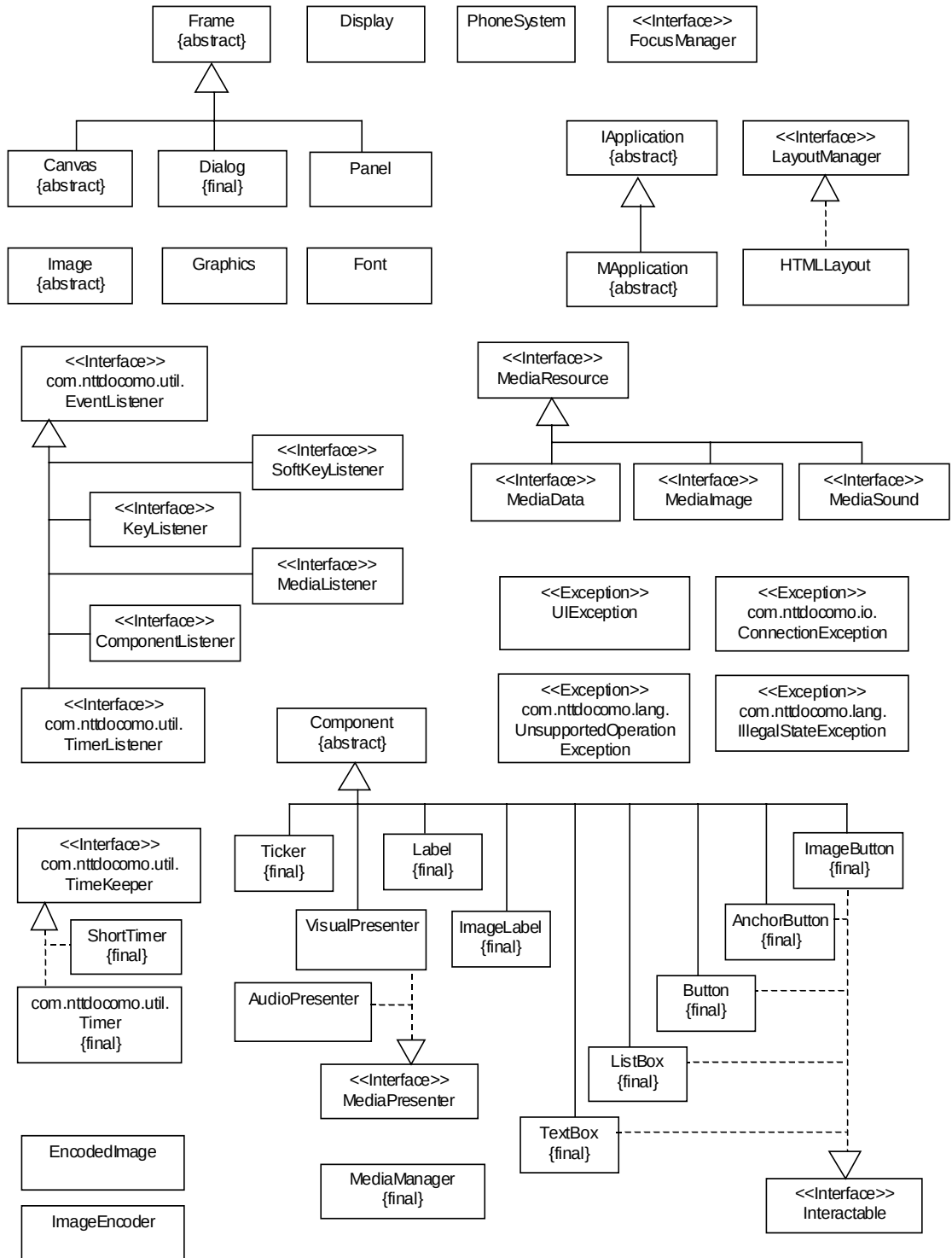


Figure 7: UI Class Hierarchy

4.1 Programming with High-level UI Components

The i-appli Runtime Environment provides content developers with nine basic UI components:

- Label
- ImageLabel
- Button
- AnchorButton
- ImageButton
- ListBox
- TextBox
- Ticker
- VisualPresenter

The other basic UI component is the `Dialog`. The behavior of these elements will be nearly the same across different manufacturers, but the look and feel may be different. This is parallel to the characteristics of standard GUI applications developed for the Windows, Macintosh, and UNIX computer platforms. The applications behave the same across the different platforms, but the look and feel is different. For example, button shapes on different platforms might appear rounded or square, and might have a thick or thin border. The same is true for different mobile phones, whose low-level graphic routines might have a different look. The default layout, scrolling, and focus management strategies will be roughly the same for all manufacturers. However, each manufacturer may choose to extend layout, scrolling, or focus management for their specific mobile phone. The i-applis Runtime Environment contains a `Component` class which defines the common characteristics of the UI components.

However, this class is prepared for the mobile phone manufacturers to provide the above basic UI components. Any developers cannot define nor use their own components inheriting the `Component` class.

[DoJa-2.0]

The components `AnchorButton` and `ImageButton` are newly added with DoJa-2.0 Profile.

4.1.1 Using the Panel

The `Panel` class acts as a simple container for other UI elements. Only one panel can be displayed on the screen at any given time. This does not restrict an i-appli to defining only one panel. Most i-applis will consist of multiple panels. The user will navigate between these panels during the normal operation of the i-appli.

It is possible to specify a title for `Panels` which indicate the corresponding function or purpose.

4.1.2 Using the Dialog

The dialog is used to display a text message to alert or inform the user, and/or prompt the user to make a decision. Dialogs are displayed on the screen overlapping the Panel or Canvas. The Dialog class extends Frame. The Dialog consists of a title, a text message, and at least one button. Only one Dialog can be displayed at a time.

The following types of Dialog are available:

- Information
- Warning
- Error
- Yes/No
- Yes/No/Cancel

Notes:

- An exception will be thrown when an attempt is made to display more than one Dialog at the same time.
- Dialogs are modal.
- Special care should be taken on the application program side so that any background threads do not interfere with the modal operation of a Dialog. For instance, the resulting behavior when a background thread attempts to change the background Panel or Canvas is device-dependent.
- Although in the same way as for Panel and Canvas, Dialog is a sub-class of the Frame class, it is not possible to display a dialog using the method `Display.setCurrent()` (i.e., the method used to change the frame being displayed on-screen). Rather, the method `Dialog.show()` is used to display the Dialog screen.

4.1.3 Using Components

There are seven UI components (subclasses of `com.nttdocomo.ui.Component`) : Label, ImageLabel, Button, AnchorButton, ImageButton, TextBox, ListBox, Ticker, and VisualPresenter. With the exception of AnchorButton, it is not possible for the size of the component to exceed the size of the screen (or in cases where a title is set for a Panel, the size of the screen not including the title area).

A Panel can be added only once (`Panel.add()` method) for one component's object.

The following section will provide a description of each UI component.

[DoJa-2.0]

With DoJa-1.0 Profile, the operation resulting from specification of a component size in excess of the screen size is depended on the manufacturer in question.

Label

The `Label` class is a component for displaying a single line of text on the `Panel`. The `Label` component does not have a border and it displays a string of characters. The text inside the `Label` component can be aligned left, center, or right. The following code sample shows the `Label` component:

Example: Label component

```
package uidemo;

import com.nttdocomo.ui.*;

/**
 * Class that illustrates the label widget and correspondingly
 * called setSize method function.
 * This class creates one button and two labels.
 * A button object is used to illustrate the
 * label widget visible/invisible feature.
 */
public class LabelDemo extends UIDemoPanel {
    Label lbl1, lbl2;
    Button btn1;
    int count = 0;

    LabelDemo() {
        lbl1 = new Label("Label1");
        lbl2 = new Label("This is a Label2");
        btn1 = new Button("Button1");
        lbl1.setSize(25,15);    // sets label size.

        this.add(btn1);
        this.add(lbl1);
        this.add(lbl2);
        ListenerClass lclass = new ListenerClass();
        this.setComponentListener(lclass);
    }
}
```

Note:

- A `Label` component cannot receive focus.

ImageLabel

The `ImageLabel` class is a component used to display images on the `Panel`. The `ImageLabel` component can only display a static image. `ImageLabels` for which no static images are set are filled in by the background color of the component. The standard Swing classes of the Java GUI can create icons by setting images to `label` components, but in the i-appli Runtime Environment, a separate component has been defined. The following code sample shows the `ImageLabel` component:

Example: ImageLabel component

```
import com.nttdocomo.ui.*;
import com.nttdocomo.io.*;

public class ImageLabelDemo extends UIDemoPanel {
    ImageLabelDemo() {
        ImageLabel ll;

        // get the media image
        MediaImage mi =
```

```

        MediaManager.getImage("resource:///UIDemo/img/cup.gif");
    try {
        mi.use();
    } catch (ConnectionException ce) {
        // Handle resource I/O exceptions here
    }
    Image im = mi.getImage();

    // create an imagelabel with an image object.
    ll = new ImageLabel(im);
    // add the imagelabel to the panel
    add(ll);
}
...
}

```

Notes:

- ImageLabel components cannot receive focus.
- In DoJa-2.0 Profile, a Graphics object extracted from Image can be used to render to an image. When using this functionality to render to the image of an ImageLabel already being displayed on the screen, the contents may not be immediately reflected by the display, depending on the phone's manufacturer. In such situations, the rendered content can be reflected to the screen by rerendering all components, such as using the ImageLabel.setImage() method to reset the image.

Button

The Button class is a component that represents buttons, and implements the Interactable interface for receiving user actions. The Button component can be set to display text. The component can be set to enabled or disabled, The following code sample shows the Button component:

Example: Button component

```

import com.nttdocomo.ui.*;

public class ButtonDemo extends UIDemoPanel implements ComponentListener {
    Button b1, b2, b3, b4;
    ButtonDemo() {
        // create four buttons and add them to the panel
        b1 = new Button("B4 enabled");
        b2 = new Button("B4 disabled");
        b3 = new Button("ChangeB4");
        b4= new Button("B4");

        this.add(b1);
        this.add(b2);
        this.add(b3);
        this.add(b4);
        ...
    }
    ...
}

```

AnchorButton

The `AnchorButton` class is a component which represents HTML anchor-type buttons (i.e., underlined text which can operate as a button), and it implements the `Interactable` interface required to receive user operations. With `AnchorButton` components, it is possible to set both text and an image to be used as a heading. If the specified text is longer than the width of the component, this text will automatically be wrapped onto several subsequent lines.

The vertical size of an `AnchorButton` is not restricted by the size of the screen in this direction; however, the horizontal size of this component is restricted to the screen's horizontal size.

This type of component can be setup as either enabled or disabled. If an `AnchorButton` has been rendered disabled, its text will not be displayed with an underline, and it will have the same look and feel as regular text. For this reason, it is also possible to use disabled `AnchorButton` components to present long text passages to the user within a `Panel`.

The following code sample shows the `AnchorButton` component.

Example: `AnchorButton` component

```
import com.nttdocomo.ui.*;

public class AnchorButtonDemo extends UIDemoPanel implements ComponentListener {
    AnchorButton anchorButton1, anchorButton2;
    AnchorButtonDemo() {
        // Create an AnchorButton capable of operation.
        anchorButton1 = new AnchorButton ("Jump to the next page");
        this.add(anchorButton1);
        // Create an AnchorButton incapable of operation, and use this
        // to present text to the user.
        anchorButton2 = new AnchorButton(
            "An AnchorButton setup to be incapable of
             operation can be" +
            "used to present long text passages to the
             user.");
        anchorButton2.setEnable(false);
        this.add(anchorButton2);
        ...
    }
    ...
}
```

Note:

- In DoJa-2.0 Profile, a `Graphics` object extracted from `Image` can be used to render to an image. When using this functionality to render to the image of an `AnchorButton` already being displayed on the screen, the contents may not be immediately reflected by the display, depending on the phone's manufacturer. In such situations, the rendered content can be reflected to the screen by re-rendering all components, using the `AnchorButton.setImage()` method to reset the image.

ImageButton

The `ImageButton` class is a component which can paste images, and it implements the `Interactable` interface required to receive user operations. An `ImageButton` for which no valid image has been set will be painted with the component's background color.

This type of component can be setup as either enabled or disabled.

The following code sample shows the `ImageButton` component.

Example: ImageButton component

```
import com.nttdocomo.ui.*;
import com.nttdocomo.io.*;

public class ImageButtonDemo extends UIDemoPanel implements ComponentListener {
    MediaImage mImage;
    Image image;
    ImageButton imageButton;
    ImageButtonDemo() {
        // Create an ImageButton and add it to the panel.
        try {
            mImage = MediaManager.getImage("resource:///label.gif");
            mImage.use();
            image = mImage.getImage();
        } catch (ConnectionException ce) {
            ...
        }
        imageButton = new ImageButton(image);
        this.add(imageButton);
        ...
    }
    ...
}
```

Note:

- In DoJa-2.0 Profile, a `Graphics` object extracted from `Image` can be used to render to an image. When using this functionality to render to the image of an `ImageButton` already being displayed on the screen, the contents may not be immediately reflected by the display, depending on the phone's manufacturer. In such situations, the rendered content can be reflected to the screen by re-rendering all components, such as using the `ImageButton.setImage()` method to reset the image.

ListBox

The `ListBox` class is a component that represents buttons, and implements the `Interactable` interface for receiving user actions. The `ListBox` component has a rectangular border, and is internally scrollable. If all of the items to be displayed in the do not fit on the screen, an indicator appears to represent vertical scrolling. The following types of `ListBox` are available:

- Single-selection list
- Multiple-selection list
- Radio button
- Check box
- Numbered list
- Option menu (pop up)

Depending on the type of `ListBox` that you see, items contained inside the frame have different appearances and selection methods, as follows:

Type	Selection	
	Single selection	Multiple selection
List	Single selection list (<code>SINGLE_SELECT</code>)	Multiple selection list (<code>MULTIPLE_SELECT</code>)
Numbered list	Numbered list (<code>NUMBERED_LIST</code>)	Not applicable
Check box	Not applicable	Check box list (<code>CHECK_BOX</code>)
Radio button	Radio button list (<code>RADIO_BUTTON</code>)	Not applicable
Pop-up	Option menu (<code>CHOICE</code>)	Not applicable

This component can be set to enabled or disabled. The following code sample shows the `ListBox` component.

Example: `ListBox` component

```
package uidemo;

import com.nttdocomo.ui.*;

/**
 * Class that demonstrates listbox widget features.
 * The list types that are demonstrated are radio_button, numbered_list,
 * checkbox, choice, single_select, and multiple select.
 */
public class ListBoxDemo extends UIDemoPanel implements SoftKeyListener {
    ListBox l1, l2, l3, l4, l5, l6;
    int count = 0;
    int newItemCnt=0;
    ListBoxDemo() {
        Label radioLabel, checkLabel, numberLabel;
        Label choiceLabel, singleLabel, multipleLabel;

        radioLabel = new Label("Radio Button List");
        // create a listbox of type RADIO_BUTTON and add some items to it.
        l1 = new ListBox(ListBox.RADIO_BUTTON);
        l1.append("radio1");
        l1.append("long radio label");
        l1.append("radio3");
        l1.append("radio4");

        checkLabel = new Label("Checkbox List");
        // Create a listbox of type CHECK_BOX, and add some items to it.
        l2 = new ListBox(ListBox.CHECK_BOX);
        l2.append("checkbox1");
        l2.append("checkbox2");
        l2.append("checkbox3");
        l2.append("checkbox4");
        l2.append("checkbox5");

        numberLabel = new Label("Number List");
        // Create a listbox of type NUMBERED_LIST, and add some items to it.
        l3 = new ListBox(ListBox.NUMBERED_LIST);
        l3.append("number1");
        l3.append("number2");
        l3.append("number3");
```

```

13.append("number4");
13.append("number5");

singleLabel = new Label("Single List");
// Create a listbox of type SINGLE_SELECT, and add some items to it.
14 = new ListBox(ListBox.SINGLE_SELECT);
14.append("single1");
14.append("single2");
14.append("single3");
14.append("single4");
14.append("single5");

multipleLabel = new Label("Multiple List");
// Create a listbox of type MULTIPLE_SELECT, and add some items to it.
15 = new ListBox(ListBox.MULTIPLE_SELECT);
15.append("multiple1");
15.append("multiple2");
15.append("multiple3");
15.append("multiple4");
15.append("multiple5");

choiceLabel = new Label("Choice List");
// Create a listbox of type CHOICE, and add some items to it.
16 = new ListBox(ListBox.CHOICE);
16.append("choice1");
16.append("choice2");
16.append("choice3");
16.append("choice4");
16.append("choice5");

this.add(radioLabel);
this.add(l1);
this.add(checkLabel);
this.add(l2);
this.add(numberLabel);
this.add(l3);
this.add(singleLabel);
this.add(l4);
this.add(multipleLabel);
this.add(l5);
this.add(choiceLabel);
this.add(l6);

//set the softkey listener on this panel
this.setSoftLabel(Frame.SOFT_KEY_2, "List");
setSoftKeyListener(this);
}
...
}

```

Notes:

- Numbered lists are a special type of single-selection list, and the user can select nine of them from the header by using the numeric keys of the mobile phone. To select the first item, use the number 1 key; to select the second item, use the number 2 key; and so on. The user selects the tenth and subsequent items using the same operations as in the single-selection list.
- Option menus (pop ups) are also a special type of single-selection list. When the mobile phone's "select" button is pressed while the option menu has focus, a pop-up appears. The user selects the desired item from this pop up. Unlike with other types of `ListBox`, the first item on the option menu starts off selected by default.
- In a `ListBox` that has been added to the current frame, when item setting operations (`setItems()`, `append()` etc.) from the application program and user operations with regards to that `ListBox` are performed simultaneously, the behavior varies depending on the manufacturer. When performing item setting operations from the application program on a `ListBox` in the current frame, use the `setEnabled()` method to disable user operations.

TextBox

The `TextBox` class is a component that represents input fields, and implements the `Interactable` interface for receiving user actions. The `TextBox` component displays single or multiple lines of text. This is the only component that accepts text input. When a `TextBox`'s edit mode is set to editing enabled by the application program, the user can launch the IME (Input Method Editor) and enter and change text, while setting it to un-editable makes the `TextBox` read-only.

The `TextBox` constructor specifies either `DISPLAY_PASSWORD` or `DISPLAY_ANY` as its display mode parameter. An i-appli requiring text input to be hidden would use a `DISPLAY_PASSWORD` mode, which displays all characters as the '*' character. If the display mode parameter is set to `DISPLAY_ANY`, text entered in the `TextBox` component is displayed as-is.

The initial input mode (e.g. kana/kanji input, alphabetical input etc.) can be changed by calling the `setInputMode()` method with the corresponding initial input mode parameter of `KANA` (kana/kanji input mode), `ALPHA` (alphabetical input mode), or `NUMBER` (numeric input mode). `TextBox` is initial set to `KANA`.

This component can be enabled or disabled. The following code sample shows the `TextBox` component.

Example: TextBox component

```
import com.nttdocomo.ui.*;

public class TextBoxDemo extends UIDemoPanel implements ComponentListener {
    TextBox t;
    TextBox t2;
    Button b;
    TextBoxDemo() {
        t2 = new TextBox(
            "This is a test for textbox", 20, 3, TextBox.DISPLAY_ANY);
        t = new TextBox(
            "This is a test for textbox", 20, 1, TextBox.DISPLAY_ANY);
        b = new Button("TestText");

        this.add(b);
        this.add(t2);
        this.add(t);
        ...
    }
    ...
}
```

Note:

- In situations where the user is editing text using the `TextBox` (i.e., when the IME is active), care must be taken with the application program to ensure that the background thread does not interfere with the IME's model operations. For example, the operations occurring when the IME is launched and when the background thread tries to change the `back Panel` to a different frame will differ from model to model; accordingly, the appropriate care should be taken.

Ticker

The `Ticker` class is a component for displaying streaming character strings on the panel. The `Ticker` component displays text in a ticker tape format. The string scrolls across the screen from right to left.

The following code sample shows the `Ticker` component.

Example: Ticker component

```
package uidemo;

import com.nttdocomo.ui.*;

// Class that demonstrates the features of the ticker widget. Initially,
// when the panel is displayed, the default string in the ticker is moving.
// The user can change the text string by clicking on the text button.
// Ticker can also be set to visible or invisible.

public class TickerDemo extends UIDemoPanel implements ComponentListener {
    Ticker ticker;          // ticker widget
    Button textButton;      // change text

    static int    whichText  = 1;
    static String textString1 = "This is a test text 1 for the ticker panel";
    static String textString2 = "Now you see test text 2";

    TickerDemo() {
        ticker      = new Ticker(textString1);
        textButton = new Button("change text");

        this.add(ticker);
        this.add(textButton);
        this.setComponentListener(this);
    }

    // Indicates that an event has occurred on Component c.
    //
    // @param c the Component on which this event has occurred
    // @param type a type of event
    // @param param parameter of event
    public void componentAction(Component c, int type, int param) {

        // A component action has occurred. If a textButton is clicked,
        // then the ticker's string will change to something different.

        if (c == textButton) {
            if (whichText==1) {
                whichText=2; ticker.setText(textString2);
            } else if (whichText==2) {
                whichText=1; ticker.setText(textString1);
            }
        }
    }
}
```

Notes:

- There is no way to control the scrolling behavior of the text inside the component from the application program.
- The `Ticker` component cannot receive focus.
- The speed of the scrolling text is manufacturer dependent.

VisualPresenter

The `VisualPresenter` class is a component for presenting dynamic images, and implements `MediaPresenter`. The `VisualPresenter` component is a high-level API component that plays back media data (e.g. GIF animations) on the screen. This component can use `MediaListener` to process events related to media data playback. For more details on event processing, see Section 4.3. For media playback code examples, see Section 4.4.

Notes:

- The `VisualPresenter` is a component, and can only be used from the high-level API.
- The `VisualPresenter` component cannot receive focus.

4.1.4 Placing Components

Developers can either leave the placement of components to the layout manager, or disable the layout manager and specify the placement position of each component using the `setLocation()` method of the `Component` class.

The layout manager is used with a setting made to `Panel`, and specifically, the method `Panel.setLayoutManager()` is used to set the layout manager in `Panel`.

Two different layout managers are provided in the i-αppli Runtime Environment — namely, the default layout manager as setup in the initial condition for `Panel` generation, and the HTML layout manager for creating layouts with a HTML-type specification method.

The default layout manager is used principally for simple, flow-type layouts. Components are added to the panel horizontally in sequence from left to right, and when no screen width remains to fit the next component, it is placed in the next available position (i.e., in the next row).

Furthermore, with the HTML layout manager, specifications such as line-break (i.e., the BR tag), new paragraph (i.e., the P tag), and alignment (i.e., the CENTER or DIV tag) can be implemented during the addition of components to `Panel`, thus enabling the developer to control the layout in a more explicit manner.

Developers should keep their component layout simple, in order to create general-purpose i-αpplis that are implementable on any manufacturer's mobile phone. Note that the number of components that can be placed will be affected by the memory utilization and the specific limitations of each device.

Notes:

- The `LayoutManager` interface is provided by a manufacturer to enable layout-manager implementation, and a developer cannot use this interface to create his or her own layout manager. The `Panel` object's `setLayoutManager()` method can be called to switch to the layout manager used by that `Panel`.
- To disable the layout manager, call the `setLayoutManager()` method with null passed as the parameter on the `Panel` object. If the layout manager is disabled, the application program uses the `setLocation()` method of each component to specify the location of each component one at a time. Note that if the layout manager is disabled, scrolling will also be disabled on some devices.
- A single HTML layout manager object cannot make settings for two or more `Panels` at the same time. If one i-αppli contains a number of `Panels` for which it is desired to apply HTML layouts, please ensure that individual HTML layout managers are allocated to each of these `Panels`.
- Whenever an HTML layout manager is set to a `Panel` using the `setLayoutManager()` method, the initial condition will be adopted (i.e., left alignment with no line-break or paragraph settings).

The following code sample shows the positioning of components using the HTML layout manager.

Example: Component placement with the HTML layout manager

```
import com.nttdocomo.ui.*;

public class HTMLLayoutDemo extends UIDemoPanel {
    Label      label1, label2, label3, label4;
    HTMLLayout  lm;

    HTMLLayoutDemo() {
        label1 = new Label("Label 1");
        label2 = new Label("Label 2");
        label3 = new Label("CENTER");
        label4 = new Label("RIGHT");

        // Generate an HTML layout manager and set to Panel.
        lm = new HTMLLayout();
        setLayoutManager(lm);

        // Specify a layout and add Label to Panel.
        add(label1);
        // All subsequent components will be placed on the next
        // line (i.e., BR tag).
        lm.br();

        add(label2);
        // All subsequent components will be placed on the next
        // line (i.e., BR tag).
        lm.br();

        // Display the components added between begin() and
        // the corresponding end() with center justification.
        lm.begin(HTMLLayout.CENTER);
        add(label3);
        lm.end();

        // Display the components added between begin() and
        // the corresponding end() with right justification.
        lm.begin(HTMLLayout.RIGHT);
        add(label4);
        lm.end();
    }
}
```

The following code sample shows the layout manager being disabled, and the application program independently placing components.

Example: component placement with setLocation()

```
import com.nttdocomo.ui.*;

public class NullLayoutDemo extends UIDemoPanel {
    Button button1, button2;
    Button offScreen;
    TextBox text;
    Label title, button1Label, button2Label;

    NullLayoutDemo() {
        // use all components here
        title      = new Label("Null Layout ");
        button1     = new Button("Button1");
        button2     = new Button("Button2");
        button1Label = new Label("ONE:");
        button2Label = new Label("TWO:");
    }
}
```

```

text                = new TextBox("A simple sentence in a
                        textbox", 40,
                        2, TextBox.DISPLAY_ANY);

/*
 * We are creating this button as an example of a
 * component that is placed off the screen
 */
offScreen = new Button("This button is never seen");

/*
 * Here is where we set the layout manager to null. This
 * means we are required to set the location of each component.
 * NOTE: components placed outside of the physical screen
 * coordinates will not be displayed at all. Scrolling may
be
    * disabled with a null layout.
    */
setLayoutManager(null);

/* NOTE: the order in which we add the components still has
 * importance. This is the order in which the components
 * will receive focus
 */
add(title);
add(button1Label);
add(button1);
add(button2Label);
add(button2);
add(text);
//Add this component too.
add(offScreen);

/*
 * Now we need to set the location of each component.
 * In other words, we do the panel layout by ourselves.
 * We can let the components set their own size, or we can
 * set the size directly by calling setSize.
 */

title.setLocation(12, 0);
button1Label.setLocation(10, 18);
button1.setLocation(45, 18);
button2Label.setLocation(10, 40);
button2.setLocation(45, 40);
text.setLocation(2, 60);

/*
 * NOTE: this button has now been placed off the screen,
 * so it is not visible and will never get focus.
 */
offScreen.setLocation(1000, 1000);
    }
}

```

[DoJa-2.0]

The HTML layout manager (i.e., HTMLLayout class) is newly added with DoJa-2.0 Profile.

4.1.5 Focus and Scrolling

The complicated behavior of focus management and scroll management often varies with respect to manufacturer. i-appli mobile phones are equipped with simple default focus managers. With the typical default focus manager, the order in which components are added to the `panel` will be the order in which the components receive focus. When an i-appli is launched and the screen is displayed, the first interactable and enabled component will have focus. However, when a component with focus is positioned outside the display range, it will not be possible to operate that component.

The default scrolling behavior is tightly integrated with focus management, because the screen scrolls to follow shifts in focus.

Note:

- The `FocusManager` interface is provided by a manufacturer to enable focus-manager implementation, and a developer cannot use this interface to create his or her own layout manager. The `Panel` object's `setFocusManager()` method can be called to switch to the focus manager used by that `Panel` object. However, no focus manager other than the default focus manager has been defined for previous profiles.

Note that unlike the layout manager, the focus manager cannot be disabled.

4.1.6 High Level Event Processing

In the screen made up by `Panel`, events generated by components are passed to the `componentAction()` method of the object implemented by the `ComponentListener`, and softkey events are passed to the `softKeyPressed()` or `softKeyReleased()` method of the object implemented by the `SoftKeyListener`.

Both the `ComponentListener` and `SoftKeyListener` are registered on the `Panel` before use. See Section 4.3 for a detailed code sample of the event listener for high-level event processing.

4.1.7 Font Support in Components

In DoJa-3.0 Profile, the `setFont()` method has been added to the `Component` and `Dialog` classes, allowing the font of the characters displayed in these to be specified from the application program. By creating a `Font` object for the font to be used and specifying it in the arguments of the `setFont()` method, fonts other than the default font can be displayed.

When a font is set to a component, this method will use it to display all character strings on that single component. Character strings composed of multiple, different fonts cannot be displayed on the same component. If this method is called on a component that has no functionality for displaying characters, nothing happens.

When a font is set to a dialog, this method will use it to display only the text body. The font for the dialog's title and buttons cannot be changed.

Note:

- The display character fonts for the panel titles and softkey labels cannot be changed.

4.2 Programming with Low-level UI API

Applications like games sometimes have a wider proliferation of graphic rendering features than interfaces using components. This section discusses how to use the low-level API and handle events, mainly for graphic rendering.

4.2.1 Using the Canvas

The `Canvas` is a screen area used for drawing graphics. Developers can subclass the `Canvas` class and override the `paint()` method to draw inside the canvas. Whenever the system determines that `Canvas` rendering is necessary or when an application program specifically requests re-rendering (i.e., when the `repaint()` method is called), the method `paint()` will be called back by the system.

As with the `Panel`, a single `Canvas` object corresponds to a single screen, so it is not possible to display more than one `Canvas` on a single screen, or display a `Canvas` and a `Panel` on the screen simultaneously. It is possible, however, to alternate between multiple `Canvases` and `Panels` on a single screen.

Following is a simple example of drawing on the `Canvas`.

Example: Drawing to the Canvas

```
import com.nttdocomo.ui.*;
public class CanvasDemo extends Canvas {
    int x = 20;
    int y = 10;

    CanvasDemo() {
        // Create soft label "black".
        setSoftLabel(Frame.SOFT_KEY_1, "Back");
    }

    /**
     * Draw string and rectangle on the canvas area.
     */
    public void paint(Graphics g) {
        g.lock();
        g.clearRect(0, 0, 1000, 1000);
        g.drawString("Hello", x, y);
        g.drawString("Canvas", x, y+15);
        g.drawRect(x+10, y+30, 20, 5);
        g.unlock(true);
    }

    ...
}
```

Notes:

- The `repaint()` method is used to specify the need for rendering to the system, and it is not possible to guarantee that a single calling of the `repaint()` method will correspond to a single calling of the `paint()` method. When considered from the application program, therefore, there will be cases where several consecutive callings of the `repaint()` method are grouped together in a single calling of the `paint()` method. Even in situations where a developer can group `paint()` methods into a single calling, it will be necessary to describe the `paint()` methods so that screen rendering may be carried out correctly.
- In DoJa-3.0 Profile, calls to the `repaint()` method on a `Canvas` that isn't set to the current frame are ignored. In profiles older than DoJa-2.x Profile, behavior depends on the manufacturer when the `repaint()` method is called on a `Canvas` that isn't set to the current frame.

4.2.2 Drawing Graphics

A graphics object (of the `Graphics` class) is used to draw lines, shapes, and text. It is also called a “graphics context,” because it contains information about the drawable area, fonts, colors, etc. The example above uses the graphics context “g” to clear the screen by using the `clearRect()` method, then drawing text by calling the `drawString()` method, and next drawing a rectangle by calling the `drawRect()` method.

The `lock()` and `unlock()` methods shown in the example above are supplied by the `Graphics` class to control the double-buffering behavior of graphics rendering. Double buffering eliminates flickering by painting into an offscreen buffer that is subsequently copied onscreen.

Some manufacturers do not support double buffering. On devices that do not support double buffering, these two methods have no effect. On devices that support double buffering, after the `lock()` method is called (for one or more times), painting to the screen will be temporarily disabled until the `unlock()` method is called once with a `true` argument, or for a matching number of times with a `false` argument.

Notes:

- Because the implementation of double buffering varies from one mobile phone to another, you must obey the following guidelines when using double buffering in order to guarantee proper application program behavior.
 - Do not intermix calls to the `lock()` and `unlock()` methods of different `Graphics` objects.
 - You must unlock a `Graphics` object before your `paint()` method returns.

The following code snippet shows correct usage of the lock/unlock methods.

Example: correct implementation (in accordance with the guidelines above)

```
import com.nttdocomo.ui.*;
public void paint(Graphics g) {
    Graphics g2 = g.copy();
    g.lock();
    // draw some stuff...
    g.unlock(false);

    g2.lock();
    // draw more stuff...
    g2.unlock(false);
}
```

The following code snippet shows incorrect usage of the lock/unlock methods.

Example: incorrect implementation

```
import com.nttdocomo.ui.*;
public void paint(Graphics g) {
    Graphics g2 = g.copy();
    g.lock();
    // draw some stuff...
    g2.lock();                // intermixing g's lock with g2's
    // draw more stuff...
    g.unlock(false);         // g2 is not unlocked
}
```

- Do not issue a `Graphics.dispose()` method for graphics objects passed as arguments of the `paint()` method. Where necessary, dispose of graphics objects passed to the `paint()` method using the Java runtime environment. Alternatively, in situations where the application program uses the `getGraphics()` method from a `Canvas` object or an `Image` object to obtain graphics objects, please use the `dispose()` method at the point at which said graphics objects are no longer needed.

- With the i-appli Runtime Environment, it is possible to set a rectangular area using the parameters of the `repaint()` method to facilitate re-rendering only within the specified rectangular area. Although it may be assumed for DoJa-1.0 Profile that certain mobile phones may not provide support for this function with the rendering device's clipping rendering capability, all mobile phones compatible with DoJa-2.0 Profile will support this function.
- With DoJa-2.0 Profile, when a `Canvas` is redisplayed as a result of the suspension or resume of an i-appli, it is guaranteed that the `paint()` method for the corresponding `Canvas` will be called once from the i-appli Runtime Environment. However, in cases where calling of this `paint()` method does not result in complete restoration of the entire screen (i.e., where the application program carries out only partial rendering in accordance with events), the degree to which the system can restore the screen will depend on the model in question.

In DoJa-3.0 Profile, when i-appli resumes, if the system can completely restore the rendered `Canvas` contents, the `paint()` method will not be called.

- The default font for the rendering of low-level text strings with DoJa-2.0 Profile is a 12-dot font, whereas with DoJa-1.0 Profile, the size of the default font varied from manufacturer to manufacturer. Note that regardless of the profile supported, support for fonts other than the default font will vary with respect to manufacturer. In general, because it is necessary to store font information in the cellular-phone's limited memory, the actual number of fonts installed is limited to a few varieties. DoJa-1.0 and DoJa-2.0 Profiles support `Font.SIZE_MEDIUM`, the default font size.

In DoJa-3.0 Profile too, in consideration of keeping application program compatibility between profiles, the default font is a 12 dot font. However, as cellular phones are now equipped with high-precision LCD devices, the default font size has been changed to support `Font.SIZE_TINY`, which has been newly added in DoJa-3.0 Profile. In DoJa-3.0 Profile, the medium font is a 24 dot font (sizes for fonts other than tiny and medium may vary depending on the manufacturer).

Due to the advance of high-precision LCD devices, the size of a font with the same number of dots appears smaller, so in designing screen-layouts, use of a larger font should be considered depending on the situation.

- Regarding graphics rendering (sprite and 3D graphics) in DoJa-2.0 Profile, features belonging to i-appli Options and i-appli Extensions were added. For information on these, refer to the "i-appli Content Developer's Guide: i-appli Options and Extensions", and the "i-appli Content Developer's Guide: Option and Extension API Reference". Among these are items that were adopted into the i-appli standard API in DoJa-3.0 Profile.
- In DoJa-3.0 Profile, rendering to the graphics object associated with a `Canvas` only occurs if that `Canvas` is set to the current frame. If the associated `Canvas` is not set to the current frame, the call to the rendering method for that graphics object will be ignored.

4.2.3 Rendering to Image Objects

Support for rendering to `Image` objects is provided with DoJa-2.0 Profile or later.

The acquisition of `Graphics` objects from `Image` objects is carried out by calling the `Image.getGraphics()` method. Rendering specification for a `Graphics` object acquired from an `Image` object is reflected in the display content of the original `Image` object. In this way, the rendered `Image` object can be set as an `ImageLabel` or an `ImageButton` to allow the results of rendering processing by the application program to also be displayed when using `Panels`.

In order to acquire a `Graphics` object from an `Image` object using the method `Image.getGraphics()`, said `Image` object must be the newest image generated by the `Image.createImage()` method. If the `getGraphics()` method is called with respect to an `Image` object acquired using the `MediaImage.getImage()` method, an `UnsupportedOperationException` will be thrown.

Note that in the initial condition, images generated by `Image.createImage()` will be painted in the same color as the default `Canvas` background color for the model in question; however, the `Image` itself has no concept of background color. In order to specify the clearing of rendering content for a `Graphics` acquired from an `Image`, rather than using the `Graphics.clearRect()` method, the `Graphics.fillRect()` method is used after first setting the appropriate color.

The following code sample shows the usage of an Image object rendered in accordance with application program specifications as an ImageLabel.

Example: Rendering to an Image object

```
import com.nttdocomo.ui.*;
public class ImageDrawDemo extends UIDemoPanel {
    ImageLabel    iLabel;
    Image         img;
    Graphics      g;

    ImageDrawDemo() {
        // Generate a new Image object and acquire a Graphics object.
        int width  = 100;
        int height = 50;
        img = Image.createImage(width, height);
        g = img.getGraphics();

        // Render the acquired Graphics object.
        g.lock();
        g.drawRect(0, 0, width-1, height-1);
        g.drawString("Sample of rendering", 5, 16);
        g.drawString("to an Image", 5, 32);
        g.drawString("using a Graphics", 5, 48);
        g.unlock(true);

        // Set the Image object ImageLabel and display on a Panel.
        iLabel = new ImageLabel(img);
        add(iLabel);
    }
}
```

Note:

- The maximum size for generation of an Image with the Image.createImage() method will correspond to the mobile phone's rendering area in terms of both width and height. Please be aware that the ability to render Images which are larger than the mobile phone's rendering area will depend on the manufacturer's specifications.

4.2.4 Color Support

A Graphics object contains information about the color of objects. In the Graphics class there are two methods that return an integer value of color:

```
getColorOfName(int name): returns an integer value of an indicated color name (Graphics constant)
getColorOfRGB(int r, int g, int b):
    returns an integer value of an indicated RGB value.
```

The method of managing colors is manufacturer dependent, and the integer values (and return values) assigned to colors in the i-appli Runtime Environment may differ from one manufacturer to the next. Developers cannot make any assumptions about the value of colors, e.g., on a black and white screen, black could have a value of 1, whereas on a color screen the value could be 255. It will be necessary for application programs to use the above-mentioned methods to convert the color definitions common for all models (i.e., names or RGB values) to color definitions specific to the model in question.

The following example shows the incorrect codes.

Example: graphics canvas

```

import com.nttdocomo.ui.*;
public class MyCanvas extends Canvas {
    ...
    public void paint(Graphics g) {
        ...
        if(g.getColorOfName(Graphics.BLACK) == 1) { // do something
            // On some manufacturers' devices, this could be true, and the code
            // between the braces would be executed, but on others' devices
            // it could be false and the code would be skipped.
        }
        ...
    }
}

```

[DoJa-2.0]

i-applis can use color-supporting pictographic chatacters (i.e., from the external character set as defined by NTT DoCoMo for i-mode services). With color-supporting pictographic chatacters, a default display color is allocated to each individual pictographic chatacter.

The `Graphics` class from DoJa-2.0 Profile or later features an additional `setPictoColorEnabled()` method, and this is used when explicitly indicating the rendering colors of `Graphics` objects to select whether to render with the pictographic chatacter's default display color or whether to render using the rendering color as specified for the `Graphics` object.

When rendering pictographic chatacters in DoJa-1.0 Profile with the rending color for `Graphics` objects being explicitly indicated, individual manufacturer specifications will determine whether rendering of the pictographic chatacters will be carried out with the default display color or whether the rendering color as specified for the `Graphics` object will be used.

Note:

- When specifying colors from an application program, it is not possible to specify a transparent color. The usage of transparent colors in the i-appli Runtime Environment is limited to those cases where transparent GIF images are being used.

4.2.5 Low-level Event Processing

The method for low-level event processing as suitable to `Canvas` usage differs from that for high-level event processing for `Panel` usage. The `Canvas` object only handles low-level events. When a developer inherits the `Canvas` class, it is possible to over-write the `processEvent()` method. This method will be invoked with the low-level event and parameters that go along with it. The following table shows valid event types to send to `Canvas`. These event types are defined by the `Display` class.

Event type	Event type value	Parameter
KEY_PRESSED_EVENT	0	Value of pressed key
KEY_RELEASED_EVENT	1	Value of released key
RESUME_VM_EVENT	4	Do not use
UPDATE_VM_EVENT	6	Do not use
TIMER_EXPIRED_EVENT	7	Timer ID of <code>ShortTimer</code> generated by timer event.

The following code example shows the implementation of the `processEvent()` method for handling low-level events.

Example: event handling implementation

```

package spaceinv;

import com.nttdocomo.ui.*;
import java.util.Random;

/**
 * This class is used to display the all-time hall-of-fame high scores
 * that have been set for a particular high score server. Scores and
 * associated names are presented in an animated way. This class will
 * switch to the next screen when user input is received, or when the
 * idle timer maintained by this class expires.
 */
public class HighScoreScreen extends Canvas implements Runnable {
    private boolean done;
    private ShortTimer timer;

    /** Create a new instance of this class */
    public HighScoreScreen() {
        ...
    }
    /** Reset this instance to its initial state. */
    public void reset(String[] list) {
        ...
    }
    /** Needed for Runnable interface. */
    public void run() {
        ...
    }
    /** Display this screen. */
    public void paint(Graphics g) {
        ...
    }

    /** Override Canvas' processEvent() for event handling. */
    public void processEvent(int type, int param) {
        if (type == Display.KEY_PRESSED_EVENT ||
            type == Display.TIMER_EXPIRED_EVENT) {
            done = true;
            timer.stop();
            timer.dispose();
            SpaceInvaders.WELCOME.reset();
            Display.setCurrent(SpaceInvaders.WELCOME);
        }
    }
}

```

[DoJa-2.0]

With DoJa-2.0 Profile or later, the sequence for the occurrence of low-level events upon the resume of an i-appli is as defined below.

- When resuming an i-appli, the `resume()` method of the i-appli's main class is called after the `RESUME_VM_EVENT` notification, which indicates the i-appli has been resumed.
- If an `UPDATE_VM_EVENT` indicating the destruction of rendering data occurs upon the resuming of an i-appli, this will occur after `RESUME_VM_EVENT`.

Note:

- The `processEvent()` method is called in response to a user action, or to the timer or some other kind of asynchronous event. When calling the `processEvent()` method, be aware that thread allocation is up to the manufacturer. For instance, some manufacturers' devices may process multiple events generated simultaneously with a single thread. Make an effort to wrap up processing inside the `processEvent()` method extremely quickly.

4.2.6 Using the IME in the Canvas

In screens using `Panel`, the IME can be launched by user `TextBox` operations and character strings entered by the user can be obtained by the i-appli. In contrast to this, in screens using `Canvas`, the IME can be launched from the application program using the APIs newly established in the DoJa-3.0 profile. Using these APIs, when using `Canvas` it is possible for the i-appli to obtain character strings input by the user in the same way as when using `Panel`.

When the `Canvas.imeOn()` method is invoked from the application program, the IME is launched. This method is non-blocking and returns to the application program when the IME has completed start-up.

When the user finishes the IME operation (by committing or cancelling), the application program receives an IME event notification. The developer can retrieve the IME processing results by overriding the `Canvas.processIMEEvent()` method.

The below code gives an example of a `Canvas` that uses the IME.

Example: Using the IME from a Canvas

```
import com.nttdocomo.ui.*;

class MainCanvas extends Canvas {
    String inputString = "";

    public void processEvent(int type, int param) {
        // Start IME with softkey 1 release
        if(type==Display.KEY_RELEASED_EVENT&&param==Display.KEY_SOFT1)
            imeOn("initial_string",TextBox.DISPLAY_ANY, TextBox.KANA);
        // Return control to the application program when the IME
        // has finished starting
    }

    public void paint(Graphics g) {
        g.lock();
        g.clearRect(0,0,200,200);
        g.drawString("InputString: '" + inputString + "'", 0, 20);
        g.unlock(true);
    }

    public void processIMEEvent(int type, String text) {
        // This method called when user IME operation finishes
        // IME event parameters are event type (committed or
        // canceled) and the input character string.
        if (type==Canvas.IME_CANCELED) inputString = "[CANCELED]";
        else if (type==Canvas.IME_COMMITTED) inputString = text;
        repaint();
    }
}
```

Notes:

- Caution is needed on the application program side to prevent background threads from interfering with the IME's modal operations when the IME is in the starting-up state. For example, note that the behavior resulting when a background thread attempts to change the rear `Canvas` to a different frame while the IME is being started varies depending on the model of phone.
- Calls to the `imeOn()` method will be ignored when the `Canvas` is not set to the current frame or a `Dialog` is being displayed.

- The `processIMEEvent()` method call is triggered by the occasion of asynchronous events which occur when the user finishes the IME operation. Note that when calling the `processIMEEvent()` method, how the thread is assigned depends on the manufacturer. For example, in one manufacturer's model, processing may occur in one thread with respect to the IME event and the other events. Please conclude handling performed within the `processIMEEvent()` method as quickly as possible.
- The `imeOn()` method cannot be called from an inactive state stand-by application.

4.3 Event Listeners

An event listener is an object registered with an event-notifying object (e.g. `Panel` or `MediaPresenter`) to listen for particular types of events. When events occur, the object constructs the proper event type and notifies the corresponding event listener of this object. There are five types of event listeners:

- `ComponentListener`
- `SoftKeyListener`
- `KeyListener`
- `MediaListener`
- `TimerListener`

There are no event listeners for `Dialogs`. Use the `Dialog.show()` method to display a dialog on the screen. The `Dialog.show()` method returns control to the application program after the user finishes interacting with the dialog. The return value of the `Dialog.show()` method indicates which button the user pressed in the dialog.

Note:

- A listener method is called in response to a user action, or the timer or some other kind of asynchronous event. When calling a listener method, be aware that thread allocation is up to the manufacturer. For instance, some manufacturers' devices may process multiple events generated simultaneously with a single thread. Make an effort to wrap up processing inside the listener method extremely quickly.

4.3.1 ComponentListener

The `ComponentListener` interface is used upon stand-by for the launching of events for UI components on the displayed `Panel`. The interface has only one method defined: `componentAction()`. To define and register the component listener:

- The class used as a listener must implement the `ComponentListener` interface and define the `componentAction()` method. When a component event takes place, the `componentAction()` method is invoked, passing to the component listener the type of event, the component in which the event took place, and any parameters needed.
- The `setComponentListener()` method must be called on the `Panel` object, and register an instance of the listener class implemented above.

The three types of events defined for the `ComponentListener` are:

- `BUTTON_PRESSED`:
Event that is issued when a `Button` is pressed. Does not use event parameters.
- `SELECTION_CHANGED`:
Event that is issued when the selected item in the `ListBox` has changed. Passes the index of the item that was selected or whose status has changed as the event parameter (If multiple items have been selected, the index of the smallest item is passed).
- `TEXT_CHANGED`:
Event that is issued when the character input of the `TextBox` is terminated. Does not use event parameters.

Next is an example of defining and using the ComponentListener as an inner class:

Example: inner class implementing ComponentListener

```
package uidemo;
import com.nttdocomo.ui.*;

/**
 * Class that demonstrates label widget and the setSize method called
 * on the label widget.
 * This class creates one button and two label widgets.
 * Button widget is used to demonstrates the visible/invisible features
 * of the label widget.
 */
public class LabelDemo extends UIDemoPanel {
    Label lbl1, lbl2;
    Button btn1;
    int count = 0;

    LabelDemo() {
        lbl1 = new Label("Label1");
        lbl2 = new Label("This is a Label2");
        btn1 = new Button("Button1");
        lbl1.setSize(25,15);    // set the size of the label.

        this.add(btn1);
        this.add(lbl1);
        this.add(lbl2);
        ListenerClass lclass = new ListenerClass();
        this.setComponentListener(lclass);
    }

    /**
     * Inner class that will handle the button events
     */
    class ListenerClass implements ComponentListener {
        /**
         * Indicates that an event has occurred on Component c.
         */
        * @param c the Component, on which this event has occurred
        * @param type a type of event
        * @param param parameter of event. */
        public void componentAction(Component c, int type, int param) {
            /**
             * A component action has occurred. If the component is button then
             * if a count is 0, then first label in this panel is set to
             * invisible. When the count is 1, then the label is set to visible.
             */
            if (c == btn1) {
                if (count == 0) {
                    lbl1.setVisible(false);
                    count++;
                }
                else {
                    lbl1.setVisible(true);
                    count=0;
                }
            }
        }
    }
}
```

Note:

- With `TextBox` and `ListBox`, the methods provided by the corresponding classes can be used to change the text set for the `TextBox` or the `ListBox` selection condition. In this way, the notification of component events upon condition changing is possible using methods and not user operations.

4.3.2 SoftKeyListener

The `SoftKeyListener` interface is used to listen for the two softkeys to be invoked. The interface has two methods defined: `softKeyPressed()` and `softKeyReleased()`. To define and register the softkey listener:

- The class used as a listener must implement the `SoftKeyListener` interface and define the `softKeyPressed()` and `softKeyReleased()` methods. When a softkey is pressed, the `softKeyPressed()` method is invoked, and when one is released, the `softKeyReleased()` method is invoked.
- The `setSoftKeyListener()` method must be called on the `Panel` object, and an instance of the listener class implemented above must be registered.

Note:

- Softkeys can also be used from a canvas. In accordance with low-level processing methods, use `Canvas.processEvent()` instead of a softkey listener (as with ordinary keystrokes, `KEY_PRESSED_EVENT` or `KEY_RELEASED_EVENT` is notified).

4.3.3 KeyListener

The `KeyListener` interface is used to listen for all keys besides the softkeys to be invoked. The interface has two methods defined: `KeyPressed()` and `KeyReleased()`. To define and register the key listener:

- The class used as a listener must implement the `KeyListener` interface, the `KeyPressed()` and `KeyReleased()` methods. When a key is pressed, the `KeyPressed()` method is invoked, and when one is released, the `KeyReleased()` method is invoked.
- The `setKeyListener()` method must be called on the `Panel` object, and an instance of the listener class implemented above must be registered.

[DoJa-2.0]

In a `Panel` object, the direction and selection keys are used for tasks such as screen scrolling or component operation. With DoJa-2.0 Profile or later, the following definitions are made for this type of key event when using high-level API.

- Direction keys (i.e., up, down, left, or right) do not generate events. These keys are generally subjected to `Panel` processing in order to facilitate scroll and focus control.
- The selection (or confirmation) key will not generate key events only if the component in focus is being presented on the display. In situations where the component in focus is not being displayed, the selection key will be subjected to `Panel` processing for the component in question.

Note that with DoJa-1.0 Profile, the decision on which key events were to be subjected to `Panel` processing under which conditions was left to the discretion of the individual manufacturers.

4.3.4 MediaListener

The `MediaListener` interface is used when media data is played, to listen for media events (`AudioPresenter` or `VisualPresenter` events). The only method that the interface defines is `mediaAction()`. To define and register the media listener:

- The class used as a listener must implement the `MediaListener` interface and define the `mediaAction()` method. When a media event occurs, this method is invoked.
- The `setMediaListener()` method must be called on the `MediaPresenter` object, and an instance of the listener class implemented above must be registered.

The six kinds of event defined for the `AudioPresenter` are:

- `AUDIO_COMPLETE`: Sound media playing complete event
- `AUDIO_PLAYING`: Sound media playing started event
- `AUDIO_STOPPED`: Sound media playing stopped event
- `AUDIO_PAUSED`: sound media data's pause event
- `AUDIO_RESTARTED`: sound media data's restart play event
- `AUDIO_SYNC`: sound media data's synchronization event

The three kinds of event defined for the `VisualPresenter` are:

- `VISUAL_COMPLETE`: Visual media playing complete event
- `VISUAL_PLAYING`: Visual media playing started event
- `VISUAL_STOPPED`: Visual media playing stopped event

For more details on media data, refer to Section 4.4.

[DoJa-2.0] [DoJa-3.0]

- Among the restart play/pause/complete events, the only media event guaranteed to occur in DoJa-1.0 Profile is `AUDIO_COMPLETE`. In DoJa-1.0 Profile, whether the other media restart play/pause/complete events are supported depends on the manufacturer. Also, media events other than the restart play/pause/complete events are not supported in DoJa-1.0 Profile.
- In DoJa-2.0 Profile, the restart play/pause/complete events are supported by all models in both `AudioPresenter` and `VisualPresenter`. In DoJa-2.0 Profile, media events other than these are included in the i-appli Optional API.
- In DoJa-3.0 Profile, all media events mentioned in this section have been adopted into the i-appli standard API.

4.3.5 TimerListener

The `TimerListener` interface is used to listen for Timer expired events. The only method that the interface defines is `timerExpired()`. To define and register the timer listener:

- The class used as a listener must implement the `TimerListener` interface and define the `timerExpired()` method. When a timer expires, this method is invoked.
- The `setListener()` method must be called on the `Timer` object, and an instance of the listener class implemented above must be registered.

Note:

- `ShortTimer` does not have a similar interface; instead, it goes through the `Canvas`' `processEvent()` method and listens for the `TIMER_EXPIRED_EVENT`.

4.4 Using Multimedia Data

4.4.1 Presenter Benefits

The i-appli Runtime Environment provides functionality for displaying images including animation and playing sound files. The classes used for playing back visuals and sounds are `VisualPresenter` and `AudioPresenter` respectively.

Notes:

- The `VisualPresenter` is a component, and must be added onto a `Panel` object on a high-level panel in order to be used. In `VisualPresenter`, GIF animation is supported as the standard visual that can be played back.
- The `AudioPresenter` is not a component, so it can be used when either the `Panel` or `Canvas` is being displayed. In `AudioPresenter`, data in Mfi format is supported as the standard sound that can be played back.
- In the i-appli development environment for DoJa-2.0 and DoJa-3.0 Profiles provided by NTT DoCoMo, `AudioPresenter` supports Mfi format and MIDI format data. For more details regarding actual development procedures for use with the i-appli Runtime Environment, refer to Chapter 13.

[DoJa-2.0]

With the i-appli Runtime Environment supplied by NTT DoCoMo which is compatible with DoJa-1.0 Profile, the use of MFi data is not possible for sound playback. Note that only the MIDI format may be used for sound files in this environment.

The following example shows how to display an image file using the `VisualPresenter` class, and how to play a sound file using the `AudioPresenter` class.

In this example, resources bundled in the i-appli's JAR file are used as the image and sound files (using a URL starting with "resource:///"). Refer to Chapter 7 for more details on resources. In this URL, not just resources, but also the HTTP URL indicating the file on the Web server and the URL indicating the position on the ScratchPad can be specified.

Example: Display an image file

```
package uidemo;

import java.util.*;
import com.nttdocomo.ui.*;
import com.nttdocomo.io.ConnectionException;

public class VisualPresenterDemo extends UIDemoPanel {

    int          px, py;    // Current location of image on screen (inside
                           // VisualPresenter)
    VisualPresenter theVP;    // Presenter

    Button        startButton;    // Button for playing GIF animation
    Button        stopButton;    // Button for stopping GIF animation

    ListBox       repeatChoice;    // Check box for specifying whether to
                                   // loop playback
    Ticker        ticker;    // Tickertape explaining key mapping

    static String instr = "Control ImagePos: Key 2-Down 8-Up 4-Left 6-Right";

    /**
     * VisualPresenterDemo constructor
     * Get needed image data, and hand off to VisualPresenter.
     */
    VisualPresenterDemo() {
```

```

// Get image data
MediaImage mi =
    MediaManager.getImage("resource:///uidemo/img/singleloop.gif");

// Can't use image data until use() method is called.
try {
    mi.use();
} catch (ConnectionException ce) {
    // Handle I/O exceptions here
} catch (UIException uie) {
    // Handle UI exceptions here
}

px    = 0;
py    = 0;

theVP    = new VisualPresenter();
theVP.setImage(mi);

ticker    = new Ticker(instr);

startButton    = new Button("start");
stopButton     = new Button("stop ");

repeatChoice  = new ListBox(ListBox.CHECK_BOX);
repeatChoice.append("Repeat");

add(theVP);
add(ticker);
add(startButton);
add(stopButton);
add(repeatChoice);

// Set media listener
theVP.setMediaListener(new MediaListenerClass());

setTitle("Status Line");

this.setComponentListener(new ListenerClass());
this.setKeyListener(new KeyListenerClass());
}

class KeyListenerClass implements KeyListener {

    /**
     * Ignore key release events
     */
    public void keyReleased(Panel p, int key) {}

    /**
     * Adjust location of image on screen in response to key press events
     *
     * @param p        Panel where event occurred
     * @param key      Value of pressed key
     */
    public void keyPressed(Panel p, int param1) {

        if (VisualPresenterDemo.this == p) {

            if (param1 == Display.KEY_8) {          // image location (bottom)
                py--;
            } else if (param1 == Display.KEY_2) { // image location (top)
                py++;
            } else if (param1 == Display.KEY_4) { // image location (left)
                px++;
            }
        }
    }
}

```

```

    } else if (param1 == Display.KEY_6) { // image location (right)
        px--;
    }

    // Set image display location by calling
    // VisualPresenter's setAttribute() method
    theVP.setAttribute(VisualPresenter.IMAGE_XPOS, px);
    theVP.setAttribute(VisualPresenter.IMAGE_YPOS, py);
}
}

class ListenerClass implements ComponentListener {

    /**
     * Control start and stop of playback due to component events
     *
     * @param c    component where event occurred
     * @param typeinterface
     * @param param    event parameter
     */
    public void componentAction(Component c, int type, int param) {
        if (c == startButton) {          // Playback starts when play
                                          // button is pressed
            theVP.play();
        } else if (c == stopButton) {     // Playback stops when
                                          // stop button is pressed
            theVP.stop();
        }
    }
}

class MediaListenerClass implements MediaListener {

    /**
     * Here is where the media listener accepts events from the VisualPresenter,
     * and as necessary implements a loop playback function here for conducting
     * playback again.
     */
    public void mediaAction(MediaPresenter source, int type, int param) {

        if (source == theVP) {
            switch (type) {

                // Handle image playback termination events here. Indicate in the
                // panel title that playback has ended, and repeat playback as
                // necessary.
                // This processing is only valid for non-looping GIF animations
                case VisualPresenter.VISUAL_COMPLETE :
                    setTitle("Complete");

                    if ((repeatChoice.isIndexSelected(0))) {
                        ((VisualPresenter) source).play();
                        setTitle("[Comp]Play again");
                    }
                    break;

                // Handle image playback initiation events here. Indicate in the
                // panel title that playback has begun.
                case VisualPresenter.VISUAL_PLAYING :
                    setTitle("Playing");
                    break;

                // Handle image playback stop events here. Indicate in the
                // panel title that playback has stopped.
                case VisualPresenter.VISUAL_STOPPED :

```

```

        setTitle("Stopped");
        break;
    }
}

/**
 * Discard (unuse() or dispose()) the MediaImage after you are finished using
 * it.
 * Discarding the image will free up resources. The MediaImage can be reused
 * here, so don't discard it yet.
 */
public void leave() {
}
}

```

Example: Play back a sound file.

```

package uidemo;

import com.nttdocomo.ui.*;
import com.nttdocomo.io.ConnectionException;

public class AudioPresenterDemo extends UIDemoPanel {
    ListBox      loopChoice;      // Loop playback flag
    Button       playButton;      // Audio playback button
    Button       stopButton;      // Audio stop button
    Label        statusLabel;     // Status display label
    AudioPresenter theAP;         // AudioPresenter
    MediaSound    theMediaSound;  // MediaSound

    /**
     * AudioPresenterDemo constructor
     * Create play and stop buttons, and check box for selecting loop playback.
     */
    AudioPresenterDemo() {

        loopChoice = new ListBox(ListBox.CHECK_BOX);
        loopChoice.append("loop");

        playButton = new Button("play");
        stopButton = new Button("stop");
        statusLabel = new Label("status line");

        theMediaSound =
            MediaManager.getSound("resource:///uidemo/img/music.mld");

        // Can't use sound data until use() method is called.
        try {
            theMediaSound.use();
        } catch (ConnectionException ce) {
            // Handle I/O exceptions here
        } catch (UIException uie) {
            // Handle UI exceptions here
        }

        theAP = AudioPresenter.getAudioPresenter();
        theAP.setSound(theMediaSound);

        this.add(playButton);
        this.add(stopButton);
        this.add(loopChoice);
        this.add(statusLabel);
        this.setComponentListener(new ListenerClass());
        theAP.setMediaListener(new MediaListenerClass());
    }
}

```

```

    }

    class ListenerClass implements ComponentListener {
        /**
         * Listener class that accepts events from buttons for controlling
         * sound data playback and stopping.
         */
        public void componentAction(Component c, int type, int param) {
            if (c == playButton) {
                theAP.play();
            } else if (c == stopButton) {
                theAP.stop();
            }
        }
    }

    class MediaListenerClass implements MediaListener {
        /**
         * Here is where the media listener accepts events from the
         * VisualPresenter, and as necessary implements a loop
         * playback function here for conducting playback again.
         */
        public void mediaAction(MediaPresenter source,int type,int param) {
            if (source == theAP) {
                switch(type) {
                    // Handle audio playback termination events here.
                    // Indicate in the status display label that playback
                    // has ended, and repeat playback as necessary.
                    case AudioPresenter.AUDIO_COMPLETE:
                        statusLabel.setText("complete");
                        if (loopChoice.isIndexSelected(0))
                            ((AudioPresenter)source).play();
                        break;

                    // Handle audio playback initiation events here.
                    // Indicate in the status display label that playback
                    // has begun.
                    case AudioPresenter.AUDIO_PLAYING:
                        statusLabel.setText("playing ");
                        break;

                    // Handle audio playback stop events here. Indicate in
                    // the status display label that playback has stopped.
                    case AudioPresenter.AUDIO_STOPPED:
                        statusLabel.setText("stopped ");
                        break;
                }
            }
        }
    }

    /**
     * If this screen is hidden, call the AudioPresenter.stop() method to stop audio
     * playback. If you are not going to use this MediaSound any longer, you can
     * call the MediaSound.unuse() or MediaSound.dispose() method (frees up each
     * type of resource)
     * Since the MediaSound could be reused here, do not call these methods yet.
     */
    public void leave() {
        theAP.stop();
    }
}

```

[DoJa-2.0]

- With DoJa-1.0 Profile, the behavior which occurs when the `stop()` method is called with respect to an `AudioPresenter` without a media data setting will depend on the manufacturer in question. In contrast, if this should occur with DoJa-2.0 Profile or later, an exception (i.e., `UIException`) will be generated. Please use the method `MediaPresenter.getMediaResource()` to determine whether or not media data has been set for a media presenter.
- With DoJa-2.0 Profile, the following specifications have been added for the playback of animated GIFs using `VisualPresenter`.
 - The minimum number of frames to allow playback is eight.
 - When replaying a loop-type animated GIF, a replay completion event is generated each time the loop is completed.
 - The maximum number of repetitions for a loop-type animated GIF is sixteen. Even in cases where looping with a larger number of repetitions is specified, replay will stop when the sixteenth loop has been completed.

[DoJa-3.0]

With DoJa-3.0 Profile, data-file byte images can be specified for retrieving media data in addition to URLs. The byte image can be given as either a byte array or in `InputStream` format.

Also in DoJa-3.0 Profile, attributes have been added to adjust the below items related to `AudioPresenter`.

- Tempo (`AudioPresenter.CHANGE_TEMPO`)
- Volume (`AudioPresenter.SET_VOLUME`)
- Key (`AudioPresenter.TRANSPOSE_KEY`)

Similar functionality is contained in DoJa-2.0 Profile i-appli Optional API (`AudioPresenter2` class), but please note that, in terms of programming, the two are not compatible.

4.4.2 Multiple Simultaneous Playback with AudioPresenter

From DoJa-2.0 Profile or later, an application program can retrieve multiple instances of `AudioPresenter` and instruct these to playback simultaneously. The behavior of this functionality differs slightly in DoJa-2.0 and DoJa-3.0 Profiles. Below, the behavior of each profile shall be explained in outline.

[DoJa-2.0]

In DoJa-2.0 Profile, the minimum number for `AudioPresenter` simultaneous playback guaranteed across all phone models is 1. In other words, there are cases where, depending on the manufacturer, multiple simultaneous playback with `AudioPresenter` cannot be performed.

When the application program instructs simultaneous playback from more presenters than is possible, a decision is made based on the value of the priority ranking attribute assigned to the presenter (`AudioPresenter.PRIORITY`). In mobile phones which can only play back 1 sound at a time, if a high-priority presenter is directed to play while a low-priority presenter is playing, the high-priority presenter will play, cutting off the low-priority presenter. When the high-priority presenter has finished playing, the low priority presenter playback will be resumed.

In mobile phones that do not support the priority attribute, presenters which begin playing later are handled as if they have higher priority than presenters which are in the process of playing. Further, behavior when playback instructions exceed the mobile phone's capabilities (number of simultaneous sounds, number of usable channels), differs depending on the manufacturer.

[DoJa-3.0]

In DoJa-3.0 Profile, the minimum number for `AudioPresenter` simultaneous playback guaranteed across all phone models is 2.

When the application program instructs simultaneous playback from more presenters than is possible, a decision is made based on the value of the priority ranking attribute assigned to the presenter (`AudioPresenter.PRIORITY`). In mobile phones which can play back 2 sounds at a time, if a high-priority presenter is directed to play while a medium-priority and a low-priority presenter are playing, the low-priority presenter is stopped, and playback of the high-priority presenter begins (the medium-priority presenter will continue playback, uninterrupted). If multiple presenters exist with priorities such that any of them could be stopped, which presenter is stopped varies depending on the manufacturer.

In mobile phones that do not support the priority attribute, if simultaneous playback from more presenters than is possible is instructed, presenters which begin playing later are handled as if they have higher priority than presenters which are in the process of playing.

If instructions are within the limits of the mobile-phone's capabilities (number of simultaneous sounds, number of usable channels), all simultaneous sound playback will be heard, regardless of presenter priority. If instructions are performed that exceed the mobile-phone's capabilities, it is possible not all sounds will be heard, even if the instructions are within the limits of the number of presenters for which simultaneous playback is possible.

Further, with regards to `AudioPresenter` simultaneous playback, the concept of ports has been introduced in DoJa-3.0 Profile. Ports are virtual playback machines, enough of which are prepared for several simultaneous playback `AudioPresenters`. Which ports individual `AudioPresenters` are assigned can be declared explicitly with an optional argument in the `AudioPresenter.getAudioPresenter()` method. It is not possible to simultaneously play an `AudioPresenter` with a port number assignment along with an `AudioPresenter` without a port assignment.

4.4.3 Audio-Synchronization Events

DoJa-3.0 Profile `AudioPresenter` supports the firing of synchronization events in sound playback. The term "synchronization event" is used to refer to media events which are notified to `MediaListener` in synchrony with the playback of a sound. The method `AudioPresenter.setAttribute()` is used to set the synchronization-event generation attribute (i.e., `AudioPresenter.SYNC_MODE`) as either ON (i.e., `AudioPresenter.ATTR_SYNC_ON`) or Off (i.e., `AudioPresenter.ATTR_SYNC_OFF`), thus allowing the generation or non-generation of these events to be controlled.

When using the synchronization event function, any one of the channels comprising the sound data is specified as the event generation source. Subsequently, synchronization events will be notified to `MediaListener` linked to the specified channel and specified note messages.

Note that the channel specified as the event generation source will also be subject to sound generation. If this is not necessary, however, specify this channel as being muted during the creation of sound data.

[DoJa-2.0] [DoJa-3.0]

Sound playback synchronization events were not supported in DoJa-1.0 Profile, and were relegated to the i-appli Optional API in DoJa-2.0 Profile. This functionality has been adopted into the i-appli standard API from DoJa-3.0 Profile or later.

4.5 Image Encoder

Functionality related to image handling (image encoding and processing) has been enhanced in DoJa-3.0 Profile. In this section, these features shall be explained in summary.

4.5.1 Image Encoder

The image encoder provides functionality for capturing the content rendered in `Image` and `Canvas`, and encoding it to a byte image of a specified format. Using this feature, the contents of the image file can be processed on `Image` or `Canvas`, then reconverted into an image file and stored in `ScratchPad` or sent to a server.

The `ImageEncoder` class is used to encode images. By specifying a graphics format name as an argument when calling the static `getEncoder()` method, the encoder object corresponding to that graphics format can be retrieved. Handling of the actual encoding is performed by using the encoder object's `encode()` method.

When the `encode()` method is called to perform image encoding, an `EncodedImage` object is returned as the result. The `EncodedImage` is a class encapsulating the encoded image data, and from this object the image's byte image can be retrieved.

The example below shows image encoding handling.

Example: Using the image encoder

```
import com.nttdocomo.ui.*;
class MainCanvas extends Canvas {

    public void paint(Graphics g) {
        // Render something to this Canvas
        ...
    }

    public void processEvent(int type, int param) {
        // Capture contents rendered on Canvas on softkey event
        if (type==Display.KEY_RELEASED_EVENT && param==Display.KEY_SOFT1) {
            // Graphics format set to JPEG
            ImageEncoder ie = ImageEncoder.getEncoder("JPEG");
            // The coordinates of this Canvas are an area 50 dots wide
            // by 50 dots tall from (0,0).
            // Capture the canvas. Results of capture are returned as an
            // EncodedImage object.
            EncodedImage ei = ie.encode(this, 0, 0, 50, 50);
            // Stream is retrieved in order to produce the image
            // byte image (in this case, JPEG).
            InputStream is = ei.getInputStream();
            ...
        }
    }
    ...
}
```

Notes:

- The commonly supported graphics format for every model of phone is JPEG (JFIF Baseline encoding). To retrieve the JPEG image encoder, specify “JPEG” as the argument for the `ImageEncoder.getEncoder()` method. Also, the size of image, both height and width, that can be encoded is dictated by the size of the mobile phone’s rendering area. Note that whether or not an image exceeding this size can be encoded varies depending on the phone’s manufacturer.
- When capturing the Canvas, that Canvas object must be set to the current frame. Also, if the IME is being started in that Canvas object, capturing cannot be performed.
- When capturing an Image, that Image object must be an object that can be called by the `getGraphics()` method. In other words, it must be an Image object that was created with the `Image.createImage()` method.
- In some cases, capture is not supported for the Canvas and Image inheritance classes that exist in the i-appli Optional and Extension APIs.

4.5.2 Pixel Operations

Pixel operations are functions for providing direct access from the application program to each pixel (picture element) arranged in the graphics context. The application program can use the following features of the `Graphics` class as pixel operations.

- Retrieval of color information for specific pixel positions in the graphics context (`getPixel()` family methods)
- Rendering dots to specific pixel positions (`setPixel()` family methods)

Pixel operation functionality was defined as an i-appli Optional API in DoJa-2.0 Profile. In DoJa-3.0 Profile, this functionality was in part enhanced and adopted into the i-appli standard API.

Notes:

- In methods for retrieving color information (`getPixel()` family methods), if out of range coordinates, or a `Graphics` object in which the `Canvas` is not set to the current frame is used, color information indicating black will be returned. This behavior has changed from the pixel operation functionality in DoJa-2.0 Profile's i-appli optional API.
- The color retrieval methods retrieve information from the back buffer and return it to the application program when double buffering is enabled. For this reason, the colors displayed on the screen may not agree with the colors retrieved by this method.

4.5.3 Image Rotation, Flipping, Enlargement and Shrinking

The `Graphics.setFlipMode()` method is called to rotate or flip images. If this method is called, specifying a rotation or flip method with respect to the `Graphics` object, a rotated/flipped image will be rendered with subsequent calls to the rendering method (`drawImage()` method or `drawScaledImage()` method).

The rotation/flip methods that can be specified are as follows.

- `Graphics.FLIP_HORIZONTAL` : image will be horizontally flipped.
- `Graphics.FLIP_NONE` : no flipping
- `Graphics.FLIP_ROTATE` : image will be flipped lengthwise and breadthwise (180 degree rotation).
- `Graphics.FLIP_ROTATE_LEFT` : image will be rotated 90 degrees left.
- `Graphics.FLIP_ROTATE_RIGHT` : image will be rotated 90 degrees right.
- `Graphics.FLIP_VERTICAL` : image will be flipped vertically.

The `Graphics.drawScaledImage()` method is called to enlarge or shrink an image. The rendering positions for the image and graphics context are set in this method's arguments, in addition to the below:

- Width and height of the rectangular area of the graphics context to which the image will be rendered.
- Upper left coordinate, and width and height of the rectangular area that is to be rendered from the rendering source image

The contents of the rendering source's rectangle will be enlarged or shrunk to exactly fit into the rendering target rectangle.

Image rotation, flipping, and enlargement/shrinking were specified as an i-appli optional API in DoJa-2.0 Profile. In DoJa-3.0 Profile, this functionality was in part enhanced and adopted into the i-appli standard API.

Chapter 5

Controlling Communications

The i-appli Runtime Environment provides Networking API, a set of classes used by application developers to communicate with Web server-side components. The objective of these classes is to make the development of Java client applications as simple as possible (see Figure 5 in Chapter 2, “Generic Connection Class Hierarchy”). In the i-appli Runtime Environment, application programs can use the HTTP(S) protocol (i.e., HEAD, GET, and POST methods) to communicate with web browsers. Standard J2SE libraries provide a rich set of functionality for handling input/output and communications with storage devices. The i-appli Runtime Environment provides basic functionality for connecting the mobile phone to a Web server based on the Generic Connection framework of the CLDC specification (subsection 2.2.2). The following sections describe how to use the HTTP(S) Connection interface to communicate with the server.

The i-appli Runtime Environment Networking APIs differ significantly from the J2SE `java.net.HttpURLConnection` in the following ways:

- There is no concept of content handler factories for plugging in handlers capable of recognizing message content type and parsing content into object form.
- We are targeting memory footprint reduction by weeding out the number of supported content types. Ordinary applications only use a limited number of content types, and so we do not employ a general framework supporting multiple content types.
- There is no generic `URLConnection` (`java.net.URLConnection` class). To keep the number of classes to a minimum, the generic `URLConnection` API is merged with the `HttpConnection` interface. This reduces memory footprint, and simplifies the API developers are to use.
- There is no `URLStreamHandlerFactory` or `URLStreamHandler` API. The i-appli Runtime Environment limits the number of available network protocols to HTTP and HTTPS, and so a general framework supporting a large number of communication protocols is not employed. This reduces memory footprint, and eliminates security issues.
- `HttpConnection` in the i-appli Execution API defines a strict call sequence for the Generic framework API. See Section 5.1 for details.

5.1 Client-Server Programming

Many programs work as client-server pairs. In the case of i-appli service, the server is on a remote host system accessible via the Internet, and the client is a Java application running on the mobile phone. A client and server typically communicate by knowing the IP address (a unique identifier for the server) and a port number, which is an identifier that allows a connection to be routed to a specific process. For simplicity, network connections are made to look like I/O streams as defined by the CLDC specification. You simply read and write data by using the usual Java stream classes and methods. For this reason, there is a method to get the input stream of a connection, and another to get the output stream. This allows the client and server to talk to each other. When the client wants an Internet service (such as retrieving a Web page from an HTTP server), it uses the URL to access the service. The i-appli Runtime Environment provides `HttpConnection` interface as a subset to `URLConnection` on J2SE. The following code shows an example of setting up and working with (an implementation object of) the `HttpConnection` interface.

Example: establish and work with a connection

```
package spaceinv;

import java.io.*;
import javax.microedition.io.*;
import com.nttdocomo.io.*;

private String[] getScoreList(int score) {
    try {
        // Retrieve the HTTP connection (HttpConnection) object.
        HttpConnection conn = (HttpConnection)Connector.open(
            "http://backflip.sfbay:8080/scores",Connector.READ_WRITE,
            true);

        // set the request method and content type.
        conn.setRequestMethod(HttpConnection.POST);
        conn.setRequestProperty("Content-Type","text/plain");

        // retrieve the output stream
        OutputStream out = conn.openOutputStream();

        // Write the data (because the connection has not yet been
        // established, the written data is temporarily placed in
        // the buffer).
        out.write("Highscore-Game: Space Invaders\n".getBytes());
        out.write(("Highscore-Name: " +
SpaceInvaders.USER.getName() +
                                "\n").getBytes());
        out.write(("Highscore-Value: " + score + "\n").getBytes());

        // close the output stream.
        out.close();

        // make the actual connection to the remote resource.
        conn.connect();

        String[] result = new String[HighScoreScreen.MAX_ENTRIES];

        // retrieve the input stream.
        InputStream in = conn.openInputStream();
        if (in == null)
            throw new IOException("high score servlet unreachable");

        // read data.
        for (int i=0; i<HighScoreScreen.MAX_ENTRIES; i++) {
            result[i] = readLine(in);
        }
    }
}
```

```

        // close input stream.
        in.close();

        // close the http connection.
        conn.close();

        return result;
    } catch (Exception x) {
        return new String[] {"Scores", "currently", "unavailable."};
    }
}

```

`HttpConnection` is an interface used for accessing network resources with the HTTP or HTTPS protocol. Instances of classes implementing this interface can be used to both read from and write to resources specified with a URL. In general, connection with the URL follows the following procedure:

1. Call the `Connector.open()` method, and create a connection object. The following correspond to the open mode and request methods:

<code>Connector.READ:</code>	corresponds to GET and HEAD methods
<code>Connector.READ_WRITE:</code>	corresponds to POST method
2. Set the setup parameter (request method and request header). Two request headers can be set from the application program: `If-Modified-Since` and `Content-Type`.
3. Call the `HttpConnection.openOutputStream()` method, and get the output stream of this connection.
4. Write to the output stream using the `OutputStream.write()` method.
5. Close the output stream using the `OutputStream.close()` method.
6. Actually connect to the remote object using the `HttpConnection.connect()` method. After this, it is possible to refer to the response header as needed. For information regarding the response headers that can be referenced from the application program, refer to the API Reference (`HttpConnection.getHeaderField()` method).
7. Get the input stream using the `HttpConnection.openInputStream()` method.
8. Read from the input stream using the `InputStream.read()` method.
9. Close the input stream using the `InputStream.close()` method.
10. Close the connection using the `HttpConnection.close()` method.

Steps 3 – 5, above, correspond to writing the HTTP request body when using the POST method. Steps 7 – 9, meanwhile, correspond to reading the HTTP response body using both the POST or GET methods.

None of these processes are compulsory and should be used whenever so required; however, when such processing is being carried out, please ensure compliance with the overall sequence as described above. For example, calling `HttpConnection.openOutputStream()` after calling `HttpConnection.connect()` could generate an exception on some devices.

A figure illustrating the state transitions from connection start to finish is shown below:

HttpConnection State Transitions

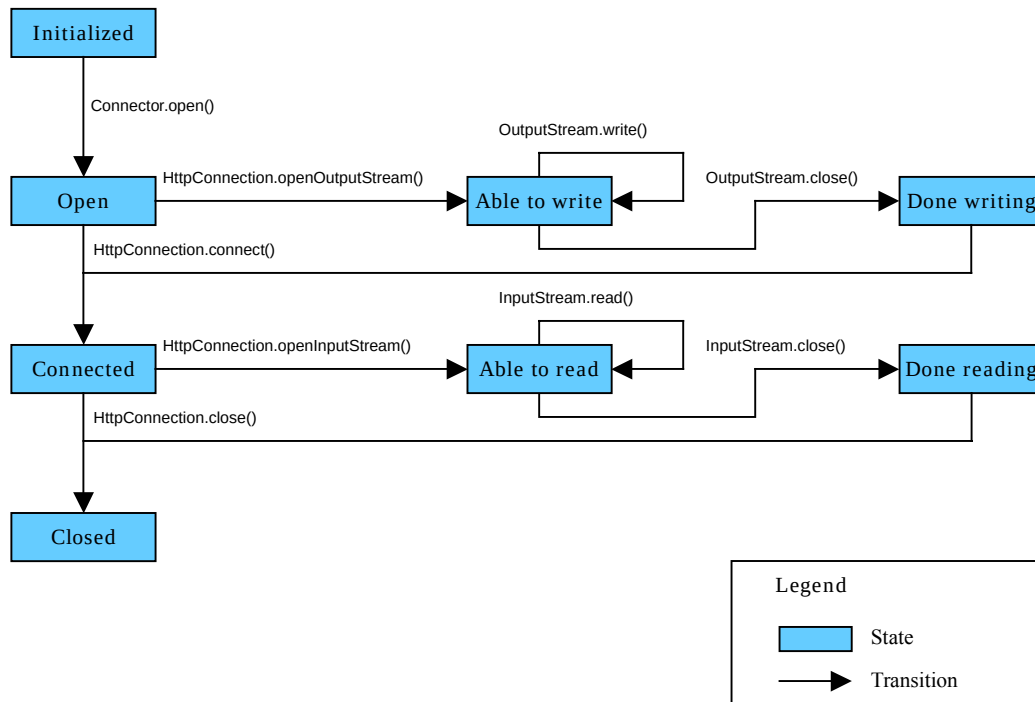


Figure 8: HttpConnection State Transitions

Notes:

- The URL passed to `HttpConnection` has the following limitations:
 - The IP address must be symbolic (e.g. "www.nttdocomo.co.jp"). `HttpConnection` cannot use numerical IP addresses like "127.0.0.1".
 - The URL protocol, host name, and port passed to `HttpConnection` must be the same as that of the URL from which the i-appli was downloaded (specified by the ADF's `PackageURL` key).
 - The multiplicity of `HttpConnection` from any one i-appli at any one time is one.
 - The i-appli service client-server system interposes an i-Mode server between the client (mobile phone) and server (Web server). The i-Mode server sometimes returns a response status as 200 when it detects one of the errors shown below. When this happens, the client side can't check for normality based on the response status. In order for the client side to conduct a strict normality check, have it make a decision based on what was added to the unique HTTP response header and HTTP response body content.
 - When the internet proxy server for the i-mode server is congested.
 - During maintenance of the public menu site (upon request from a public menu site content provider)
- Note that if a response status is returned indicated an error, the i-appli Execution will throw an exception.
- Use the following utilities to encode and decode HTTP request/response bodies to and from x-www-form-urlencoded format:
 - `com.nttdocomo.net.URLEncoder`
 - `com.nttdocomo.net.URLDecoder`
 - The sizes of messages that can be sent and received using HTTP(S) communications are limited as follows:
 - Size of requests that can be sent: 10 Kbytes
 - Size of responses that can be received: 20 Kbytes

These restrictions apply not only to situations where application programs are performing direct `HttpConnection` operations, but also when media data is acquired through HTTP communication.

- Sometimes, the URL accessed by the i-appli will have a BASIC authentication setting on the server side. When this type of URL is accessed, the mobile phone will automatically display a dialog prompting the user to enter a user ID and password. The application program cannot independently send BASIC authentication information (e.g. an `Authorization` request header) to the server.
- The i-appli will respond to an incoming voice call during communication in accordance with the mobile phone's i-Mode incoming call setting.
- When sending an HTTP request to a Web server using the POST method, be sure to specify an appropriate content type in the HTTP request body settings. Set the content type (`Content-Type` header) using the `HttpConnection.setRequestProperty()` method. Furthermore, if there is no data to be written into the request body, the GET method is to be used instead of the POST method. When using the former, please ensure that no setting is made for the content type.
- For i-mode service, it is imperative that a content-length setting (i.e., `Content-Length` header) be made in the response from the web server. Also in the case of HTTP(S) communication from an i-appli, please be sure to set the content length in the web server's response.
- If an i-appli is suspended (see item 3.6) while it is carrying out data communication, this communication will also be suspended. In such a case, an exception will be generated as soon as the i-appli is restarted.

[DoJa-2.0]

- Mobile phones which are compatible with DoJa-2.0 Profile feature a Self Mode function which terminates all external communication and allows usage only of functions which do not carry out such communication. If the user sets the mobile phone to Self Mode, the launching of i-applis will be possible, but it will not be possible for these i-applis to make use of HTTP(S) communication.
- In terms of the URL lengths specified for HTTP(S) communication with DoJa-2.0 Profile or later, settings of up to 255 bytes are possible for scheme and pass, and up to 512 bytes for queries.
- With DoJa-1.0 Profile, the following definitions are made with regard to status codes for HTTP response.

– `HttpConnection.HTTP_SERVER_ERROR (500)`

– `HttpConnection.HTTP_INTERNAL_ERROR (501)`

In contrast, the following are defined for DoJa-2.0 Profile or later.

– `HttpConnection.HTTP_INTERNAL_ERROR (500)`

– `HttpConnection.HTTP_NOT_IMPLEMENTED (501)`

Note that in order to guarantee compatibility upon compilation, the definition of `HttpConnection.HTTP_SERVER_ERROR (500)` is continued in DoJa-2.0 Profile.

[DoJa-3.0]

Message size restrictions for sending and receiving with HTTP(S) communication have been relaxed in DoJa-3.0. In profiles prior to DoJa-2.0 Profile, the transmission size in the request was limited to 5KB, and the reception size in the response was limited to 10KB.

5.1.1 Terminating the Connection

Close the connection between the server and client by using the `close()` method of the `HttpConnection` interface. However, this method will not free the actual physical resource until all I/O streams obtained from `HttpConnection` have been closed.

Note that with DoJa-2.0 Profile or later, if the `connect()` method is in the communicating condition, the `close()` method can be called for that `HttpConnection` instance from other threads so that communication can be terminated.

Note:

- You need to explicitly close EVERY connection (implementation object of the `Connection` interface) that you retrieve by calling `Connector.open()`, and EVERY I/O stream. Note that this also applies to situations where exceptions are generated for communication errors and the like as a result of the use of `Connection` implementation objects acquired with the `Connector.open()` method. Failing to do so will cause a resource leak, and may prevent you from opening a new connection.

5.2 Session Management

The HTTP protocol does not maintain a long-term connection. A connection is established only when client needs to exchange data with a server. On the other hand, user verification is required on servers which permit usage by specified users only; however, it is not practical to login to servers from user each time communication is made. In order to facilitate the repeated exchange of data with servers by a client, it will be necessary to maintain the condition where the user remains logged into the system.

With the Session Management method, when a user logs onto the system, the session manager creates a string (session ID) uniquely identifying that session. Then after this, the server and client exchange session IDs to identify each HTTP connection owner (logged-on user). In order to maintain security, session IDs are invalidated after a set amount of time has passed without any access. Owners of invalidated session IDs have to go back and log on to use the system again.

In general, cookies are widely used for the actual session management implementation. Cookies maintain a session between the client and server by setting user information – in this case, the session ID – in specific HTTP request/response headers. In the i-αpli Runtime Environment, however, the only information that can be appended to an HTTP request header is Content-Type and If-Modified-Since, so cookies cannot be used. In the i-αpli Runtime Environment, set the session ID in the request's target URL or HTTP body instead of using cookies.

In the following example, session IDs are exchanged using HTTP request/response bodies instead of cookies.

Example: Sample session management implementation

```
package banking;

import java.io.*;
import javax.microedition.io.*;
import com.nttdocomo.io.HttpConnection;

/**
 * This class implements a communication process with the bank server.
 */
public class BankingClient {
    private static final String UNAUTHORIZED =
        "verification of account data failed";

    private String url;
    private String cookie;
    private HttpConnection conn;
    private InputStream in;

    /**
     * Create an instance of this class. At this time, create
     * an initial connection to the server to verify the supplied ID and password.
     * @param acct account name of the user
     * @param pin account password of the user
     * @exception IOException will be thrown if the server is unreachable
     * or if the supplied ID and password are rejected.
     */
    public BankingClient(String url, String acct, String pin)
        throws IOException {
        this.url = url;
        conn = (HttpConnection)Connector.open(url,
            Connector.READ_WRITE, true);
        conn.setRequestMethod(HttpConnection.POST);
        conn.setRequestProperty("Content-Type", "text/plain");
        OutputStream out = conn.openOutputStream();
```

```

// Set the account information in the HTTP request body.
out.write((VERIFY + " " + acct + " " + pin).getBytes());
out.close();

conn.connect();

in = conn.openInputStream();
if (getCookie(readLine(in)) == null) {
    throw new IllegalArgumentException(UNAUTHORIZED);
}

}

private String getCookie(String line) {
    String header = "Cookie: ";
    // Obtain the session ID from the
    // row read from the HTTP response body
    if (!line.startsWith(header) || line.length() <= header.length())
        return null;
    return line.substring(header.length());
}

private String contactServer(String cmd) {
    try {
        // Close any old connections
        if (in != null) in.close();
        if (conn != null) conn.close();

        // re-open connection to server, and set the session
        // ID obtained in the initial connection made
        // from the constructor of this instance in the HTTP
        // request body.
        conn = (HttpURLConnection)Connector.open(url,
            Connector.READ_WRITE, true);
        conn.setRequestMethod(HttpURLConnection.POST);
        conn.setRequestProperty("Content-Type", "text/plain");
        OutputStream out = conn.getOutputStream();
        out.write((cmd + "\n").getBytes());
        out.write(("Cookie: " + cookie + "\n").getBytes());
        out.close();

        conn.connect();

        in = conn.openInputStream();
        if (in == null) {
            throw new IOException(UNAUTHORIZED);
        }

        return readLine(in);

        // do not close input stream and connection, as
        // subsequent processes will retrieve additional
        // data from the connection.
        // Connection will be closed next time this method is
        // called, or by a call to the close() method of this
        // class.
    } catch (IOException ioe) {
        return null;
    }
}
}

```

5.3 Secure Communication Using HTTPS

Client server applications that require secure connections will use the HTTPS protocol. The client program code for connecting to an HTTPS server will be exactly the same as for the HTTP protocol, except that the protocol session in the URL string will be set to “https” instead of “http.” For example, to communicate the Space Invaders high scores and names between client and server securely, the only change needed is the URL string. In other words, if a service using HTTP protocol is provided at Port No. 8080 on the server (i.e., `backflip.sfbay`), and if a similar service using HTTPS protocol is provided at Port No. 7070, then the URL: `http://backflip.sfbay:8080/` is to be changed to `https://backflip.sfbay:7070/`.

Notes:

- Due to URL restrictions requiring URLs passed to `URLConnection` having the same protocol, host, and port number as the URL the i-appli was downloaded from originally, only i-applis that were downloaded over https may use https. This may also mean that an i-appli downloaded using https will not be able to use the HTTP protocol.
- Web servers using HTTPS communications are required to install a server certification containing the signature of an i-appli service Certification Authority. i-appli service supports the following root Certification Authority certificates.
 - 1) Verisign
 - Verisign class 3 Primary CA root certificate
 - Verisign class 3 Primary root certificate G2
 - RSA secure server root certificate
 - 2) Baltimore
 - GTE CyberTrust Root CA
 - GTE CyberTrust Global Root

[DoJa-2.0]

With DoJa-2.0 Profile or later, if SSL data for a verification error or the like is generated during the execution of SSL communication, `ConnectionException(SSL_ERROR)` will be thrown. The exception status `SSL_ERROR` has been added to DoJa-2.0 Profile.

Chapter 6

Text Processing

In general application text handling, in addition to the ASCII character set, the character set specific to each country is often used. The iMode service, employs Shift-JIS as the encoding method for Japanese text, but the i-appli runtime environment uses the CLDC internationalized stream input/output structure, making it simple to handle Shift-JIS Japanese text.

6.1 Text Processing

The Java Virtual Machine performs all internal text processing in Unicode. For this reason, any text data encoded in a format other than Unicode must be converted to Unicode to become a Java character stream. CLDC possesses an internationalized stream input/output structure similar to that of J2SE, and classes are provided by the `java.io` package for character code conversion between Unicode character streams and other text byte streams. With the `InputStreamReader` class, you can convert byte streams to character streams. You use the `OutputStreamWriter` class to translate character streams into byte streams. Next Figure illustrates the conversion process:

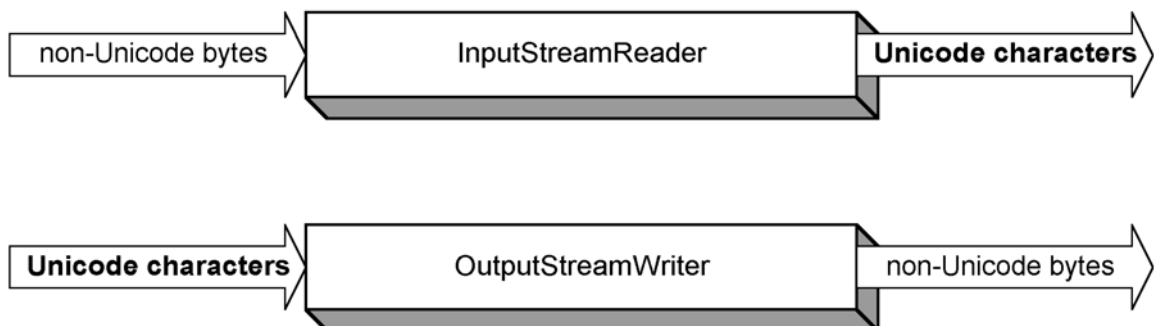


Figure 9: I/O Stream Conversion

The application program must use the `InputStreamReader` and `OutputStreamWrite` classes to ensure proper conversion of input and output text from the network. The iMode service employs Shift-JIS as the encoding method for Japanese text, and the default encoding for these classes is SJIS.

The following code sample shows the conversion methods.

Example: Conversion of SJIS byte data into Unicode characters.

```
java.io.*;
import com.nttdocomo.io.*;
import javax.microedition.io.*;

conn = (HttpConnection)Connector.open(url, Connector.READ_WRITE, true);

// Convert the SJIS byte data into Unicode characters
isr = new InputStreamReader(conn.openInputStream()); //default encoding is SJIS
```

Example: Conversion of Unicode characters into SJIS byte data.

```
java.io.*;
import com.nttdocomo.io.*;
import javax.microedition.io.*;

conn = (HttpConnection)Connector.open(url, Connector.READ_WRITE, true);

// Convert the Unicode characters into SJIS byte data.
osw = new OutputStreamWriter(conn.openOutputStream()); //default encoding is SJIS
```

[DoJa-3.0]

The following two items have been added in DoJa-3.0 Profile's `com.nttdocomo.io` package:

- Buffered reader (`BufferedReader`)
- Writer for formatting output (`PrintWriter`)

These classes offer functionality corresponding to a subset of the classes of the same names contained in the J2SE `java.io` package.

6.2 i-Mode Pictographic Characters for i-appli Service

Pictographic characters (i-Mode external character set) supported by traditional i-Mode content can also be used in i-appli service. Below is a code table (SJIS/Unicode matrix) showing the pictographics characters defined by NTT DoCoMo, and installed on every mobile phone. Codes use hexadecimal notation.

Be aware that the font displayed in the table is slightly different from the one actually installed on mobile phones. In general, the font installed on actual mobile phones is more grainy, due to limitations on the number of dots available for drawing characters.

Basic pictographic characters

Char	SJIS	Unicode	Title	Char	SJIS	Unicode	Title
	F89F	E63E	Fine		F8CB	E66A	Convenience store
	F8A0	E63F	Cloudy		F8CC	E66B	Gas station
	F8A1	E640	Rain		F8CD	E66C	Parking
	F8A2	E641	Snow		F8CE	E66D	Traffic light
	F8A3	E642	Lightning		F8CF	E66E	Bathroom
	F8A4	E643	Typhoon		F8D0	E66F	Restaurant
	F8A5	E644	Fog		F8D1	E670	Coffee shop
	F8A6	E645	Drizzle		F8D2	E671	Bar
	F8A7	E646	Aries		F8D3	E672	Beer
	F8A8	E647	Taurus		F8D4	E673	Fast food
	F8A9	E648	Gemini		F8D5	E674	Boutique
	F8AA	E649	Cancer		F8D6	E675	Beauty salon
	F8AB	E64A	Leo		F8D7	E676	Karaoke
	F8AC	E64B	Virgo		F8D8	E677	Movie
	F8AD	E64C	Libra		F8D9	E678	Right upward diagonal
	F8AE	E64D	Scorpio		F8DA	E679	Amusement park
	F8AF	E64E	Sagittarius		F8DB	E67A	Music
	F8B0	E64F	Capricorn		F8DC	E67B	Art
	F8B1	E650	Aquarius		F8DD	E67C	Play
	F8B2	E651	Pisces		F8DE	E67D	Event
	F8B3	E652	Sports		F8DF	E67E	Ticket
	F8B4	E653	Baseball		F8E0	E67F	Smoking
	F8B5	E654	Golf		F8E1	E680	No smoking
	F8B6	E655	Tennis		F8E2	E681	Camera
	F8B7	E656	Soccer		F8E3	E682	Bag/tote
	F8B8	E657	Skiing		F8E4	E683	Book
	F8B9	E658	Basketball		F8E5	E684	Ribbon
	F8BA	E659	Motor Sports		F8E6	E685	Present
	F8BB	E65A	Pager		F8E7	E686	Birthday
	F8BC	E65B	Train		F8E8	E687	Telephone
	F8BD	E65C	Subway		F8E9	E688	Mobile phone
	F8BE	E65D	Bullet train		F8EA	E689	Memo
	F8BF	E65E	Car (sedan)		F8EB	E68A	TV
	F8C0	E65F	Car (SUV)		F8EC	E68B	Game
	F8C1	E660	Bus		F8ED	E68C	CD
	F8C2	E661	Boat		F8EE	E68D	heart
	F8C3	E662	Airplane		F8EF	E68E	Spade
	F8C4	E663	House		F8F0	E68F	Diamond
	F8C5	E664	Building		F8F1	E690	Club
	F8C6	E665	Post office		F8F2	E691	Eye
	F8C7	E666	Hospital		F8F3	E692	Ear
	F8C8	E667	Bank		F8F4	E693	Hand (rock)
	F8C9	E668	ATM		F8F5	E694	Hand (scissors)
	F8CA	E669	Hotel		F8F6	E695	Hand (paper)

Char	SJIS	Unicode	Title	Char	SJIS	Unicode	Title
	F8F7	E696	Right downward diagonal		F986	E6E1	Mobile Q
	F8F8	E697	Left upward diagonal		F987	E6E2	1
	F8F9	E698	Foot		F988	E6E3	2
	F8FA	E699	Shoe		F989	E6E4	3
	F8FB	E69A	Glasses		F98A	E6E5	4
	F8FC	E69B	Wheelchair		F98B	E6E6	5
	F940	E69C	New moon		F98C	E6E7	6
	F941	E69D	Quarter moon		F98D	E6E8	7
	F942	E69E	Half moon		F98E	E6E9	8
	F943	E69F	Crescent moon		F98F	E6EA	9
	F944	E6A0	Full moon		F990	E6EB	0
	F945	E6A1	Dog		F991	E6EC	Black heart
	F946	E6A2	Cat		F992	E6ED	Fluttering heart
	F947	E6A3	Resort		F993	E6EE	Broken heart
	F948	E6A4	Christmas		F994	E6EF	Hearts (plural)
	F949	E6A5	Left downward diagonal		F995	E6F0	Yeah! (happy face)
	F950	E6AC	Action!		F996	E6F1	Grr! (angry face)
	F951	E6AD	Bag		F996	E6F1	Grr! (angry face)
	F952	E6AE	Pen		F997	E6F2	Aww! (disappointed face)
	F955	E6B1	Doll		F998	E6F3	Sniffle (sad face)
	F956	E6B2	Chair		F999	E6F4	Dizzy
	F957	E6B3	Night		F99A	E6F5	Good (upward arrow)
	F95B	E6B7	soon		F99B	E6F6	Music
	F95C	E6B8	on		F99C	E6F7	Feelin' good (hot spring)
	F95D	E6B9	end		F99D	E6F8	Cute
	F95E	E6BA	Clock		F99E	E6F9	Kiss mark
	F972	E6CE	phone to		F99F	E6FA	Shiny (new)
	F973	E6CF	mail to		F9A0	E6FB	Idea
	F974	E6D0	fax to		F9A1	E6FC	Disgust (anger)
	F975	E6D1	I-Mode		F9A2	E6FD	Punch
	F976	E6D2	I-Mode (w/frame)		F9A3	E6FE	Bomb
	F977	E6D3	Email		F9A4	E6FF	Atmosphere
	F978	E6D4	Provided by DoCoMo		F9A5	E700	Bad (downward arrow)
	F979	E6D5	DoCoMo point		F9A6	E701	Sleepy (asleep)
	F97A	E6D6	Fee service		F9A7	E702	Exclamation
	F97B	E6D7	Free		F9A8	E703	Exclamation & question
	F97C	E6D8	ID		F9A9	E704	2 exclamations
	F97D	E6D9	Password		F9AA	E705	Wham! (collision)
	F97E	E6DA	More...		F9AB	E706	Sweat beads (flying off face)
	F980	E6DB	Clear		F9AC	E707	Phew! (sweat beads)
	F981	E6DC	Search (find)		F9AD	E708	Dash (running)
	F982	E6DD	NEW		F9AE	E709	— (long vowel symbol 1)
	F983	D6DE	Position information		F9AF	E70A	— (long vowel symbol 2)
	F984	D6DF	Free dial (toll free)		F9B0	E70B	Select
	F985	E6E0	Dial using pound key				

Extended pictographic characters

Char	SJIS	Unicode	Title	Char	SJIS	Unicode	Title
	F9B1	E70C	i-appli		F9D7	E732	Trademark
	F9B2	E70D	i-appli (bordered)		F9D8	E733	Running man
	F9B3	E70E	T-shirt (border)		F9D9	E734	Top secret
	F9B4	E70F	Purse		F9DA	E735	Recycle
	F9B5	E710	Make-up		F9DB	E736	Registered trademark
	F9B6	E711	Jeans		F9DC	E737	Danger / Warning
	F9B7	E712	Snowboard		F9DD	E738	Prohibited
	F9B8	E713	Chapel		F9DE	E739	Vacant room, seat, parking space
	F9B9	E714	Door		F9DF	E73A	Pass mark
	F9BA	E715	Money bag		F9E0	E73B	No vacant room, seat, parking space
	F9BB	E716	PC		F9E1	E73C	Arrow left/right
	F9BC	E717	Love letter		F9E2	E73D	Arrow up/down
	F9BD	E718	Wrench		F9E3	E73E	School
	F9BE	E719	Pencil		F9E4	E73F	Wave
	F9BF	E71A	Crown		F9E5	E740	Mount Fuji
	F9C0	E71B	Ring		F9E6	E741	Clover
	F9C1	E71C	Stopwatch		F9E7	E742	Cherry
	F9C2	E71D	Bicycle		F9E8	E743	Tulip
	F9C3	E71E	Teacup		F9E9	E744	Banana
	F9C4	E71F	Wrist watch		F9EA	E745	Apple
	F9C5	E720	Thinking face		F9EB	E746	Flower bud
	F9C6	E721	Relieved face		F9EC	E747	Maple
	F9C7	E722	Cold sweat		F9ED	E748	Cherry blossom
	F9C8	E723	Cold sweat 2		F9EE	E749	Rice ball
	F9C9	E724	Puffy face		F9EF	E74A	Short cake
	F9CA	E725	Dumb face		F9F0	E74B	Sake bottle and cup
	F9CB	E726	Heart-shaped eyes		F9F1	E74C	Rice bowl
	F9CC	E727	Finger-gesture OK		F9F2	E74D	Bread
	F9CD	E728	Sticking out tongue		F9F3	E74E	Snail
	F9CE	E729	Wink		F9F4	E74F	Chick
	F9CF	E72A	Happy face		F9F5	E750	Penguin
	F9D0	E72B	Enduring face		F9F6	E751	Fish
	F9D1	E72C	Cat 2		F9F7	E752	Delicious!
	F9D2	E72D	Crying face		F9F8	E753	Laughter
	F9D3	E72E	Tears		F9F9	E754	Horse
	F9D4	E72F	No good		F9FA	E755	Pig
	F9D5	E730	Clip		F9FB	E756	Wine glass
	F9D6	E731	Copyright		F9FC	E757	Disheartened

Note:

- In mobile phones which do not support extended pictographic characters, these will be displayed as spaces or the equivalent. The actual character that will be displayed will depend on the model in question.

Chapter 7

Using the ScratchPad and Resources

This chapter describes how to use the ScratchPad and resources accessible from an i-appli Service mobile phone.

The ScratchPad is a storage area on mobile phones used by i-applis to store persistent data. Resources are data files (e.g. image and sound files) bundled in a JAR file.

As with the discussion of HTTP communications in Chapter 5, both the ScratchPad and resources are accessed using the Generic Connection framework.

7.1 Reading from and Writing to the ScratchPad

i-appli uses its server or the ScratchPad as the primary location for persistent data (e.g. sensitive data and data that needs to be stored for a long period of time). In the i-appli Runtime Environment, a special storage area known as a ScratchPad is assigned to each i-appli so that the storing of data can be continued even after completion of the i-appli, and that this data can be used again when the i-appli is subsequently launched. The limitations on the size of the ScratchPad available are described in Section 3.2. Also note that the only i-appli that can access a ScratchPad is the one that created it.

If an i-appli wishes to use the ScratchPad it must request it by specifying the ScratchPad size using the `SPsize` key in the ADF. The JAM will combine the ScratchPad size with the overall size of the JAR file to determine if there is enough room to download the i-appli. See Section 13.5.1 for details on how to write an ADF.

The ScratchPad memory model implements a portion of the transaction concept. Any changes you make to the ScratchPad will either be written in full or totally rejected.

The JAM creates and manages storage for each i-appli whose ADF specifies a ScratchPad size. The points at which the JAM actually writes the data to storage will depend on the optimization of the manufacturer's memory and system model.

The ScratchPad is accessed using the Generic Connection framework.

The format for ScratchPad URLs is as follows:

```
scratchpad:///<number> [;pos=<start-point> [,length=<access-length>] ]
```

A maximum of 16 ScratchPads assigned to a single application can be segment-managed by specifying an ADF `SPsize` key. In `<number>`, which ScratchPad will be accessed is specified with numbers 0 — 15.

Further, `<start-point>` value in the `pos` option specifies the access start offset in the specified ScratchPad, while the `length` option value `<access-length>` specifies a number of bytes from the access start offset. When the `pos` option is specified, data will be read or written beginning from that position. When the `length` option is specified with the `pos` option, the length of the stream will be the length specified in the `length` option (the `length` option cannot be specified without also specifying a `pos` option). As a result of applying the `pos` and `length` options, no value can be specified which falls outside of the ScratchPad area (when using segment-management, the area of the individual segments).

If the ScratchPad URL is specified to `Connector.open()`, this method returns an implementation instance of `StreamConnection`. The application program can then access the ScratchPad by retrieving I/O streams from this `StreamConnection` instance.

[DoJa-3.0]

ScratchPad segment management and the `length` option were added in DoJa-3.0 Profile. In profiles preceding DoJa-2.0 Profile, the `<number>` value specified in the ScratchPad URL will always be 0. See Chapter 13.5.1 for how to declare an ADF (SPsize key) to segment-manage the ScratchPad.

Depending on the phone model, when reading media data from the ScratchPad (i.e. when specifying a ScratchPad URL in the media data retrieval method of the `MediaManager` class) in some cases the application's execution performance can be improved by setting an appropriate `length` in the URL.

The following two examples show how to read from and write to the ScratchPad.

Example: writing to the ScratchPad area

```
import java.io.*;
import javax.microedition.io.*;

public void save() {
    try {
        OutputStream out = Connector.openOutputStream("scratchpad:///0")
        out.write((hi_score >> 24) & 0xff);
        out.write((hi_score >> 16) & 0xff);
        out.write((hi_score >> 8) & 0xff);
        out.write(hi_score & 0xff);
        out.close();
    } catch (IOException e) {
        // handle ScratchPad I/O exceptions
    }
}
```

Example: Reading from the ScratchPad area

```
import java.io.*;
import javax.microedition.io.*;

public void load() {
    try {
        InputStream in= Connector.openInputStream("scratchpad:///0")
        hi_score = in.read() << 24;
        hi_score |= (in.read() << 16);
        hi_score |= (in.read() << 8);
        hi_score |= in.read();
        in.close();
    } catch (IOException e) {
        // handle ScratchPad I/O exceptions
    }
}
```

Example: reading from the ScratchPad area

```
// retrieve the UTF (i.e. Unicode) string from the ScratchPad
// starting at the 4th byte in the ScratchPad. The first four bytes are used to store
// the score as shown in the previous example

package spaceinv;

import java.io.*;
import javax.microedition.io.*;

public class UserNameScreen extends Panel implements TimerListener {
    private String name;

    /** Create a new instance of this class. */
    public UserNameScreen() {
        try {
            DataInputStream in = Connector.openDataInputStream(
                "scratchpad:///0;pos=4");
            name = in.readUTF();
            in.close();
        } catch (Exception x) {
            // handle ScratchPad I/O exceptions
        }
        ...
    }
    ...
}
```

Notes:

- If a `StreamConnection` is opened to a ScratchPad, then `Connector.READ`, `Connector.WRITE`, or `Connector.READ_WRITE` may be specified for the open mode. Either input streams or output streams may be acquired from a ScratchPad which has been opened in `Connector.READ_WRITE` mode. However, it is not possible with a single `StreamConnection` for both the input stream and the output stream to be active at the same time. When one of these is opened, ensure that the other is set to the closed condition.

[DoJa-2.0]

ScratchPad opening with `Connector.READ_WRITE` mode is possible only with DoJa-2.0 Profile or later. In the case of DoJa-1.0 Profile, the operations which result when a ScratchPad is opened with `Connector.READ_WRITE` mode will depend on the manufacturer's specifications.

- DoJa-3.0 Profile supports Jar inflater functionality (`com.nttdocomo.util.JarInflater` class) in utilities, and entries of files compressed in Jar format can be expanded in i-applis. When giving `JarInflater` a Jar file from ScratchPad as an input stream, strictly specify the Jar file storage location using the `pos` option and `length` option in the ScratchPad's URL. If this is not done, a file format error exception may be thrown.

In the example below, the contents of a Jar file stored on the ScratchPad are read using the Jar inflater feature.

Example: Using a Jar file stored in the ScratchPad

```
// Using a Jar file 1867 bytes long, beginning at the 1K byte
// position from the front of the ScratchPad
try {
    String url = "scratchpad:///0;pos=1024,length=1867";

    // Open the read stream from ScratchPad.
    InputStream is = Connector.openInputStream(url);
    // Create Jar inflater object.
    JarInflater ji = new JarInflater(is);
    // After creating the Jar inflater object, the input stream
    // specified in the constructor is not needed.
    is.close();

    // Specify the the entry name of the file contained in the Jar
    // file and open the input stream to access the target file.
    InputStream is2 = ji.getInputStream("datafile.bin");
    for (;;) {
        // Process the input data.
        ...
    }
    // Close the Jar entry input stream.
    is2.close();
    // Close the Jar inflater.
    ji.close();
} catch (IOException ioe) {
    // This exception occurs if there was a ScratchPad access error.
    ...
} catch (JarFormatException jfe) {
    // This exception occurs if a file format error was detected
    // in the Jar inflater.
    ...
}
```

Please refer to the API reference for the details on the JarInflater class.

7.2 Reading Resources

Image, sound, and other data files bundled in the JAR file are called resources. You can access resources using the Generic Connection framework. Resources can only be opened as READ only. Explicitly request the READ only mode as a parameter to the `Connector.open()` method, or use the convenience API `Connector.openInputStream()` to obtain an input stream without obtaining a `Connection` object. If one of these is not used, an exception will be thrown. The static method `Connector.open()` accepts three parameters:

- 1) The URL you want to connect to
- 2) The open mode
- 3) Timeout exception request flag (true or false)

When accessing a resource, the first parameter should be a URL string with the format `"resource:/// <jar-relative-path>"`. The `<jar-relative-path>` portion specifies a relative path to the location in the JAR file in which the resource is stored. Set the second parameter to `Connector.READ` (because resources are READ only). Ordinarily, set the third parameter to false, unless opening an HTTP(S) connection. Do not use the short version of the `open()` method (the one that only takes the URL as a parameter), since the default mode is `Connector.READ_WRITE`.

The resource URL can be the URL string used to access the media data. This allows the application to easily play back GIF and sound files bundled in the JAR file.

The following example shows a resource being used as media data. In this example, the application retrieves a file called "bank.gif" that is bundled in the JAR file, and sets it to the `VisualPresenter`.

Example: using a resource

```
package banking;

import com.nttdocomo.io.ConnectionException;
import com.nttdocomo.ui.*;

...
/** present the welcome screen. */
class WelcomePanel extends Panel {

    /** create a new instance of this class. */
    public WelcomePanel() {
        final Panel instance = this;
        MediaImage bank;

        VisualPresenter vp1 = new VisualPresenter();
        try {

            bank = MediaManager.getImage(
                "resource:///BankingDemo/img/bank.gif");
            bank.use();
        } catch (ConnectionException ce) {

            // handle resource access exceptions

        }

        vp1.setImage(bank);
        ...
    }
    ...
}
```

7.3 Error Handling

i-applis that utilize the ScratchPad and resource URLs should handle the following error exceptions:

ConnectionNotFoundException

Will be thrown if the URL is invalid. Invalid means that the i-appli Runtime Environment cannot parse the specified URL format, or that the URL type and open mode are in conflict.

ConnectionException

Status: SCRATCHPAD_OVERSIZE

Will be thrown if an attempt is made to write past the end of the ScratchPad.

Status: ILLEGAL_STATE

Will be thrown if a method call violates the state of the connection object. A method call violating the state of a connection object means that (for instance) the `openOutputStream()` method was called on a connection object that was created with `READ` mode.

Status: RESOURCE_BUSY

Will be thrown if more than one thread tries to open a ScratchPad or resource at the same time.

For other error conditions, an `IOException` or a sub-class of `IOException` will be thrown.

[DoJa-2.0]

With DoJa-1.0 Profile, it is specified that an `EOFException` should occur when reading is carried out beyond the end of the ScratchPad. In contrast, even if this should occur with DoJa-2.0 Profile or later, operations will conform with those for each stream class's methods as defined for CLDC. For example, the value `"-1"` will be returned if the `InputStream.read()` method is instructed to read a byte at a position beyond the end of the ScratchPad.

Chapter 8

Accessing Platform Resource

The `PhoneSystem` class defined in the `com.nttdocomo.ui` package is provided in order to control the mobile phone's platform resources from an i-appli. Contained within the `PhoneSystem` class are the `setAttribute()` method for setting attribute values for platform resources, the `getAttribute()` method for referencing the attribute values of platform resources, and the `isAvailable()` method for testing whether a specific platform resource can be controlled on the model in question. In order to control platform resources, it will be necessary to specify the platform resource type and the corresponding attribute value(s). The following types of platform resources can be specified in the i-appli Runtime Environment.

- Screen back-light (i.e., switching on or off)
- Vibrator (i.e., switching on or off)
- Phone folding condition (i.e., reference of attribute value only)
- Mail or message receipt condition (i.e., reference of attribute value only)
- Battery level (reference of attribute value only)
- Signal strength (reference of attribute value only)
- Setting status of Manner Mode (reference of attribute value only)

All of these attributes are defined in the `PhoneSystem` class.

Furthermore, the `PhoneSystem` class also contains the method `playSound()` which facilitates the playback of effect sounds that have already been included in the mobile phone. Although the different types of effect sound are defined in the `PhoneSystem` class, the sounds which may actually be played will depend on the manufacturer in question.

[DoJa-2.0] [DoJa-3.0]

Each platform's resource implementation profile and placement are as below. In the chart, "Standard" indicates the resource is located in the i-appli standard API, while "Option" indicates that it's located in i-appli Options. "No" indicates that the feature is not available in that profile.

Platform	DoJa-1.0	DoJa-2.0	DoJa-3.0
Display Backlight	Standard	Standard	Standard
Vibration	No	Option	Standard
Phone folded status	No	Option	Standard (*1)
Mail/messages received status	No	Standard	Standard
Battery Level	No	No	Standard
Signal Strength	No	No	Standard
Status of Manner Mode setting	No	No	Standard
Sound Effect	No	Option	Standard

(*1) Phone folded status is only enabled in folding or flip-type cellular phones.

Notes:

- Mail/messages received status, Battery Level, Signal Strength and status of Manner Mode setting operate the various icons shown on the cellular-phone's display. For i-appli to reference this information, it must be declared that the i-appli will reference the cellular-phone's icon information in the ADF `GetSysInfo` key, and the user must grant permission for i-appli to reference the icon information. Refer to Section 13.5.1 for more information regarding ADF notation methods.
- Please also be aware that as a result of linking of the mail and message receipt condition with the mobile phone's icon for mail and message receipt, cases may occur where the notification of mail and messages may not be received as a result of signal conditions, and in this type of situation, the desired effect may not be achieved.
- When i-applis use the screen back-light, the vibration, or any other platform resource with a high level of power consumption for extended periods of time, the system on certain models may automatically turn off the mobile phone in order to conserve power. In the platform resource condition as returned by the method `PhoneSystem.getAttribute()`, the results of automatic condition switching by such a function will also be reflected.
- The backlight cannot be controlled from inactive-state stand-by applications.
- In situations where a mobile phone's key operations are deactivated due to phone settings, it will not be possible for an i-appli to operate the vibrator.

Chapter 9

Stand-By Applications

DoJa-2.0 Profile or later provides support for stand-by application functionality, thus allowing the calendars, images, and other similar items displayed on the mobile phone's stand-by screen to be replaced with resident system i-applis which execute to provide display content. With DoJa-1.0 Profile, in order to satisfy the desire to make i-applis resident, it was necessary to be solved by which i-applis would be capable of continuous execution. However, the method was characterized by a number of different problems which resulted in the ease-of-operation of the i-appli and of the mobile phone itself being compromised. As a solution to these problems, the stand-by application functionality from DoJa-2.0 Profile or later uses a mechanism which realizes easy-to-use resident applications.

9.1 Overview of Stand-By Applications

9.1.1 Characteristics of Stand-By Applications

When the creation of resident applications using stand-by application functionality is compared with the creation of resident applications within the range of DoJa-1.0 Profile functions, the following advantages can be identified.

Improved ease-of-use

In the condition where normal i-applis are executing, the mobile phone's key operations are received as i-appli events. For example, if the number key was pressed on a mobile phone in the stand-by condition, the user-interface mechanism for the native operating system (hereinafter the "native OS") would capture the event and would then switch automatically to a mode for voice call. However, during the execution of an i-appli, all number-key operations are forwarded to the i-appli as key events. Accordingly, if users of resident applications created using conventional methods desired to make a voice call or to use a web browser, it was necessary to manually stop the current application. Furthermore, when the voice call or web-browsing was completed, the user had to immediately and manually restart the i-appli.

In contrast to this situation, stand-by application functionality allows users to make simple key operations which switch between the receipt mode for key events by i-applis (i.e., active mode) and the receipt mode for key events by the mobile phone's native OS (see the following pages). Generally, users implement stand-by applications in a manner where said i-applis are made resident in the receipt mode for key events by the native OS, and where mode switching is carried out whenever necessary to allow interaction with the i-appli.

Note that in a condition where applications are resident, if users launch voice call, a web browser, or some other type of native function, the stand-by application will be automatically suspended or ended. Furthermore, when the user operation is completed and the mobile phone subsequently returns to the stand-by condition, the stand-by application will either be automatically restarted or launched once again.

Reduced power consumption

In situations where normal i-applis are executed or where stand-by applications are in the active mode, the execution time for this application will be directly linked to the mobile phone's power consumption, and problems will thus exist with regard to running over extended periods. In addition to the active mode, the following two modes are also available so that resident applications capable of extended operation may be designed.

- Inactive mode: Continued processing by application programs is made possible simply by not notifying these i-applis of key events.
- Sleep mode: Power consumption is reduced to minimum by not notifying i-applis of key events and by terminating (i.e., sleeping) processing by application programs.

Although the execution of application program is terminated in the sleep mode, the inactive mode will be automatically readopted upon the occurrence of certain system events (i.e., the elapsing of a specific period of time, etc.), thus allowing the application program to be executed once again.

Real-time mail and message receipt

In situations where normal i-applis are executing and the receipt of mail or messages is notified, priority will lie with the execution of the i-appli, and thus the mail or messages will not be received. For this reason, the receipt of mail or messages was often not noticed by users of resident applications created using conventional methods.

In the same way during execution of normal i-applis, if stand-by applications are operating in the active mode and the receipt of mail or messages is notified, actual reception will not be carried out. However, if in either the inactive or sleep mode, receipt processing will be carried out (although execution of the application will be terminated at that time).

On mobile phones which are compatible with DoJa-2.0 Profile or later, an on-screen mark or the equivalent will provide notification of the receipt of mail or messages even during execution of normal i-applis or when in the active mode. In this, real-time user notification has been realized.

The following type of stock-price alarm has been conceived as a sample stand-by application:

1. In preparation, the user selects a number of stocks of interest and also sets price conditions so that alarms can be activated for each of these stocks.
2. Normally at a rate of once per hour, the application program obtains the newest prices for the selected stocks from its server, and when a stock over the condition price is identified, the application notifies the user with a sound or vibrator-based alarm.
3. Upon receipt of this alarm, the user can interact with the i-appli through key operations so that more information can be obtained for the stock in question.

From the above example, the processing for step 2 can be realized in the inactive and sleep modes. The application program carries out communication with the server and screen updating while in the inactive mode, and it repeatedly enters the sleep mode at hourly intervals. While this processing is being carried out, all key operations performed by the user are relayed to the native OS, and for this reason, voice-call mode can be entered simply by pressing a number key.

Furthermore, the processing for step 3 is realized by switching the application to the active mode in response to user operations. Since the user's key operations are relayed to the i-appli in this mode, processing can be carried out to allow interaction between the user and the i-appli.

Notes:

- Mode-switching key operations by the user are only supported with regard to transition from either the inactive or sleep mode to active mode. All other mode transitions are to be carried out through the calling of methods from applications.
- Assignment of the mode-switching key will vary from manufacturer to manufacturer.

9.1.2 User Settings for Stand-By Applications

As with normal i-applis, stand-by applications may be launched through manual operations by the user, or they can be registered as stand-by applications in the mobile phone so that they may become resident in the system. The execution of such system-resident stand-by applications will be suspended or ended in situations where voice-call, a web browser, or some other native function is launched as a result of user operations. Furthermore, when the user has finished with the function in question and the mobile phone returns once again to stand-by status, the i-appli will be automatically restarted or launched once again. This type of launch format is given the name “stand-by launching” to distinguish it from the launching of normal i-applis.

The following section details the general sequence to be followed from downloading of a stand-by application to setting of stand-by conditions and actual stand-by launching. Note that this represents an ideal sequence, and the details of cellular-phone operations will differ from manufacturer to manufacturer.

1. Downloading stand-by applications

Stand-by applications are downloaded using the same procedure as for normal i-applis. If the link for i-appli download on the browser is clicked, the system will first of all check the ADF content, and if the corresponding i-appli is determined to be suitable for installation on the mobile phone in question, the JAR file will be downloaded. (For more details, refer to Section 1.4.1.)

Note that the ADF downloaded in this way will contain a declaration (i.e., MyConcierge key) that its i-appli is actually a stand-by application. For more details regarding the MyConcierge key and the ADF, refer to Section 13.5.1.

2. Application settings

After the i-appli has been downloaded, application settings will be carried out in accordance with the content of the ADF. Specifically, stand-by settings, network permission settings, and settings for the reference of the icon data condition are included. If the ADF declares that its application is a stand-by application, it will also be possible for the user to perform stand-by settings.

In order to enable the stand-by launching of a stand-by application, it will be necessary to register the i-appli in the mobile phone using stand-by settings. Note that it is not possible to enable stand-by launching of multiple i-applis at any one time; however, users may modify the content of the stand-by settings made here at a later time.

3. Return to stand-by condition and launching of stand-by applications

After performing stand-by settings for an i-appli in accordance with Step 2 above, if the mobile phone is returned to stand-by condition through, for example, operation of the user's hang-up key, stand-by launching will be carried out automatically for the stand-by application.

Notes:

- In contrast to the term “stand-by launching”, the term “normal launching” is used to refer to all types of launching other than stand-by launching. In particular, the following cases correspond to normal launching:
 - i-appli is launched by a direct i-appli launch instruction from the user, such as an i-appli list menu operation.
 - When an i-appli is launched using the ADF LaunchAt key. (See Section 13.5.1)
 - i-appli is launched from another application through application linking (see Chapter 11).
 - An i-appli which launches immediately after download is automatically launched directly after downloading.

9.2 Creating Stand-By Applications

In terms of the creation of stand-by applications, this section will point out the differences which exist with the creation of normal i-applis, and it will also detail the relevant precautions.

9.2.1 Stand-By Application Class

In stand-by applications, the i-appli's main class (i.e., the class specified by the ADF's `AppClass` key) is created by inheriting `MApplication`. The `MApplication` class is a class inheriting `com.nttdocomo.ui.IApplication`, and it provides methods which, in addition to controlling the life cycle of normal i-applis, also control the complicated life cycle of stand-by applications.

The `MApplication` class can, in addition to those methods from the `IApplication` class, also make use of the following methods:

- Mode transition control
 - `deactivate()`: Implements transition from the active mode to the inactive mode.
 - `sleep()`: Implements transition from the inactive mode to the dormant mode.
- System event control
 - `setWakeupTimer()`: Starts the wake-up timer
 - `getWakeupTimer()`: Acquires the start-up timer's remaining time
 - `resetWakeupTimer()`: Cancels the wake-up timer
 - `setClockTick()`: Sets whether or not clock events will be generated
 - `processSystemEvent()`: Handler method for the processing of system events.

i-applis which inherit `MApplication` are not restricted to stand-by launching; rather, they can be launched in the same way as normal i-applis by, for example, manual user operations or timer launching. In such a case, however, it is not possible for the application program to call the corresponding methods. If an i-appli which has not been subjected to stand-by launching attempts to call one of these methods, an exemption will occur.

An explanation of the life cycle and mode transitions for stand-by applications is provided in Section 9.2.2, and an explanation of system events is provided in Section 9.2.3.

9.2.2 Life Cycle and Mode Transitions

This section will provide an explanation of the three types of mode that stand-by applications can adopt.

1. Active mode

This mode is similar to that when a normal i-appli is launched, and it allows the receipt of key events and soft-key events from stand-by applications. Since the processing of soft-key events is entrusted to the application, the display of soft-key labels is also carried out on the stand-by application side.

2. Inactive mode

Although the stand-by application continues to execute and the screen display is rendered in accordance with the stand-by application's instructions, key events and soft-key events are not notified to the stand-by application, and are instead notified to the native OS. Since the processing of soft-key events is entrusted to the native OS in such a case, it is not possible for the stand-by application to carry out the setting of soft-key labels.

A stand-by application will be in the inactive state directly after launching in stand-by mode.

3. Dormant mode

In this mode, execution of the stand-by application is stopped in order to conserve battery power. The screen will retain the rendering content from immediately before the stand-by application was switched to this mode. It is only possible perform transition to this mode from the inactive mode — that is to say, transition from the active mode is not possible.

Notes:

- Although key events are notified to the native OS in the inactive mode, screen display will be entrusted to the application. Accordingly, if an application in the inactive mode provides screen content which prompts key operations by the user, there is a possibility that incorrect key operations may be made. In order to create stand-by applications which will be easy to use, it will be necessary to give careful consideration to the screen display in the inactive mode.
- During i-appli execution, a system dialog (JAM native component) rather than the application program's will sometimes be displayed.
 - During HTTP communication, and ID and password are requested for BASIC authentication
 - During HTTPS communication, the server's certification is determined to be not-completely safe and it is necessary to confirm continued operation with the user.
 - The application program calls an API with features that require user verification to be executed, such as application linking functionality.

It will not be possible for system to provide screen content when the stand-by application is in the inactive mode, and please be aware that an exception will be generated in such a case.

During execution of a stand-by application, transition between these three modes will take place as a result of factors such as the calling of mode-transition control methods as described in Section 9.2.1 or the occurrence of system events.

The following diagram illustrates the format for mode transitions between the active, inactive, and dormant modes.

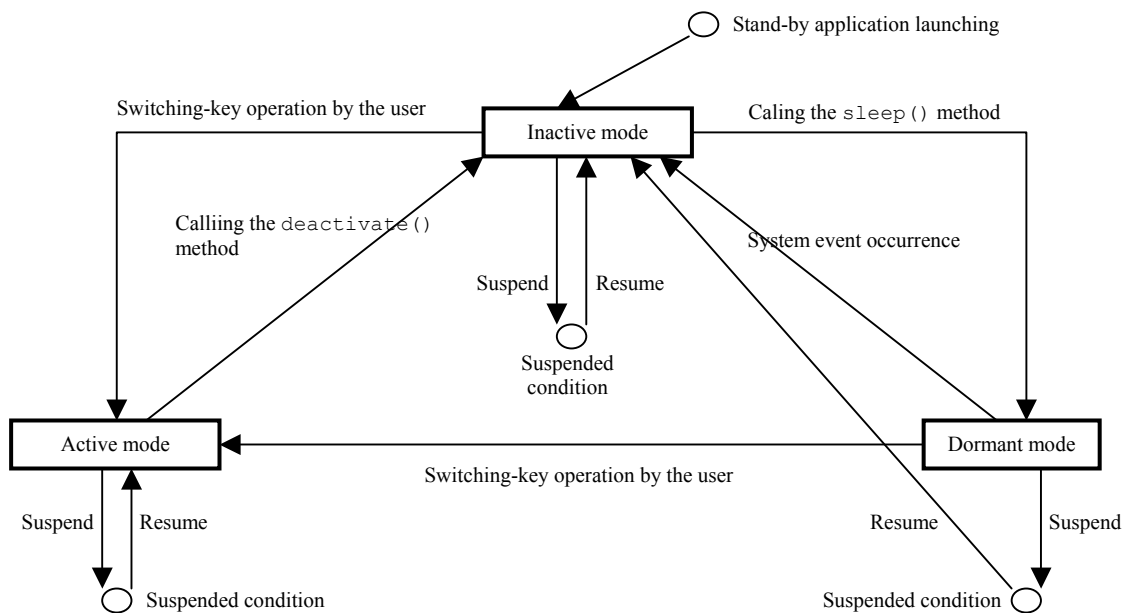


Figure 10: Transition Between Modes for Stand-By Applications

The following section will provide a description of the mode transitions between the active, inactive, and dormant modes.

1. Transition from the inactive to active mode

If, during execution of a stand-by application in the inactive mode, the user performs a switching-key operation, then transition to the active mode will occur. At this time, the system event `MODE_CHANGED_EVENT` will be notified to the stand-by application.

2. Transition from the active to the inactive mode

When a stand-by application executing in the active mode calls the method `MApplication.deactivate()`, transition to the inactive mode will occur. Nothing will occur if this method is called by a stand-by application executing in the inactive mode.

Note that although switching-key operations when in the inactive mode or the dormant mode are not notified to the stand-by application as key events, they are notified as key events (i.e., `KEY_IAPP`) when in the active mode. If the application program is set up in such a way that the `MApplication.deactivate()` method is called in response to the receipt of such a key event, the user will be able to use the switching key to toggle between modes. In such a case, ensure that calling of the `MApplication.deactivate()` method is carried out after first receiving the `KEY_IAPP` release event. If this precaution is not observed, then in the case of certain manufacturers, the system will receive the release event which occurs after the press event, thus resulting in the active mode being adopted once again.

3. Transition from the inactive mode to the dormant mode

A stand-by application which is executing in the inactive mode can call the `MApplication.sleep()` method to initiate transition to the dormant mode. If the event queue contains any unprocessed system events at the time of transition, these will be discarded. Stand-by applications which adopt the dormant mode will remain stopped until the occurrence of a system event. The thread which called the `sleep()` method will return from this method upon the occurrence of such an event, and the subsequent logic will be started.

If it is desired to return to the inactive mode after a specific period of time has elapsed, the `MApplication.setWakeupTimer()` method can be called in advance of the `sleep()` method. Alternatively, if it is desired to return to the inactive mode based on changes in the native clock's time, it will be necessary to use the `MApplication.setClockTick()` method in advance to activate clock events.

Note that all stand-by application threads will be stopped upon the adoption of the dormant mode. If the dormant mode is adopted during event processing, rendering methods (i.e., `Canvas.paint()`), or any other threads which may run asynchronously, be aware that these threads will be stopped. If the application program enters the dormant state during communications handling, that communications handling will fail.

4. Transition from the dormant mode to the active mode

When a stand-by application is executing in the dormant mode, if a switching-key operation is carried out by the user, then transition to the active mode will occur. At this time, the system event `MODE_CHANGED_EVENT` and `FOLD_CHANGED_EVENT` will be notified to the stand-by application. The thread which called the `sleep()` method will return from this method upon the occurrence of transition from the dormant to the active mode, and processing for the subsequent logic will be started. Return from the `sleep()` method will take place in advance of `MODE_CHANGED_EVENT` notification.

For more information regarding system events, refer to Section 9.2.3.

5. Transition from the dormant mode the inactive mode

If a system event other than `MODE_CHANGED_EVENT` occurs while an application is executing in the dormant mode, then transition to the inactive mode will take place. The thread which called the `sleep()` method will return from this method upon the occurrence of transition from the dormant to the inactive mode, and processing for the subsequent logic will be started. Return from the `sleep()` method will take place in advance of system-event notification

For more information regarding system events, refer to Section 9.2.3.

Transition from the active mode to the dormant mode is not possible, and an exception will occur if the `sleep()` method is called by a stand-by activation executing in the active mode. If it is desired to transfer from the active mode to the dormant mode, the `deactivate()` method should first be used to initiate transition to the inactive mode, and then the `sleep()` method should be called.

It is necessary for stand-by applications to be able to identify the mode in which they are currently executing. Appropriately manage the application's own status when there are opportunities for the `MODE_CHANGED_EVENT` to be notified or to call the `deactivate()` method.

The suspension of stand-by applications in each of these modes will occur in the following situations. As with the restart of normal i-applis, the `MApplication.resume()` method will be called upon the restarting of a suspended stand-by application.

Mode	Stand-by application is interrupted when:
Active Mode	The mobile phone receives an incoming voice call; i.e. an event with higher priority than normal i-appli execution occurs.
	That stand-by application calls an application linking method, other than the <code>MApplication.launch()</code> method, which requires a user operation (confirmation dialog operation, etc.). (A method of this kind cannot be called from the inactive mode).
Inactive Mode Dormant Mode	The mobile phone receives incoming voice calls, mail or messages, alarm notification from the schedule alarm occurs; i.e. an event with higher priority than inactive/dormant i-appli execution occurs.
	An outgoing voice call is performed by the user.
	The user launches an application that does not use packet communications. (*1)

(*1) By “applications that do not use packet communication” the native scheduler and calculator applications that are loaded onto the mobile phone and have no packet communication functionality (packet communication is not possible) are indicated.

If a stand-by application is executing in either the active or inactive mode upon suspension, then it will return to the same mode after restarting. In contrast, if a stand-by application is executing in the dormant mode upon suspension, its mode will change to inactive after restart, and the appropriate care should be taken. If a stand-by application executing in the dormant mode is suspended and restarted, then the thread which is being blocked by the `MApplication.sleep()` method will return from the `sleep()` method and the subsequent processing will be restarted.

Furthermore, stand-by applications executing in any mode will be stopped in the following situations. When a stopped application is returned to stand-by mode, it will be automatically restarted.

Mode	Stand-by application ends when:
Active	<code>MAApplication.terminate()</code> is called from the stand-by application
	That stand-by application performs application linking with the <code>MAApplication.launch()</code> method. (*1)
	User operation resulting in a forced termination.
	An external device is connected with a connection cable.
	A non-caught exception or error occurs within the stand-by application (*2) (*3)
Inactive	<code>MAApplication.terminate()</code> is called from the stand-by application
	Forced shutdown of the executing stand-by application is indicated by user menu operations
	User operation resulting in the launch of an application that uses packet communication. (*4)
	An external device is connected with a connection cable.
	A non-caught exception or error occurs within the stand-by application (*2) (*3)
	iMode and key operations have been locked from a user menu operation. In this case, stand-by applications will not be launched in stand-by until the lock is removed.
Dormant	Forced shutdown of the executing stand-by application is indicated by user menu operations
	User operation resulting in the launch of an application that uses packet communication. (*4)
	An external device is connected with a connection cable.
	iMode and key operations have been locked from a user menu operation. In this case, stand-by applications will not be launched in stand-by until the lock is removed.

(*1) Excludes cases in which the `MAApplication.launch()` method attempts to launch the browser. Stand-by applications cannot launch the browser.

(*2) When a stand-by application terminates due to an uncaught exception/error, the i-appli runtime environment will ask the user whether or not to restart that application in stand-by. If the user chooses not to launch the application, that mobile-phone's stand-by application settings will be erased. It is necessary for the stand-by application to appropriately handle any exceptions that occur during its life-cycle. (However, if the stand-by application commits a serious security violation, that mobile-phone's stand-by settings will be erased without user confirmation.)
Also, in DoJa-2.0 Profile, when a stand-by application terminates due to an uncaught exception/error, that mobile-phone's stand-by settings will be erased without user confirmation.

(*3) When a stand-by application in the inactive mode terminates due to an uncaught exception/error, the mobile-phone logs information indicating which stand-by application terminated and when. The user can reference this log from the mobile-phone's menu, etc.

(*4) By "applications that use packet communication" the browser, mailer, i-appli etc. that have packet communication functionality (packet communication is possible) are indicated. Concerning the browser or mailer launch by the user, there are mobile phone models which do not terminate, but suspend a standby application in the inactive or dormant mode depending upon the hardware capability (installed memory capacity and such).

As shown in the table, executing stand-by applications can be ended by a variety of external factors. Ensure that when data which should be stored is generated, it is immediately saved to either the ScratchPad or the server.

9.2.3 System Events

Special events (i.e., system events) used to control life-cycles can be handled within stand-by applications. Notification of system events is provided using the `processSystemEvent()` class for each instance of the stand-by application's main class (i.e., the `MApplication` sub-class). By over-writing this method, it is possible for a stand-by application to implement the processing of system events. Furthermore, this method's first parameter provides notification of the type of occurring event, whereas the second parameter provides notification of the event parameters associated with said occurring event.

The different types of system events are described below. The `MApplication` class defines constants which indicate the types for these events.

- `MODE_CHANGED_EVENT`

This event indicates that a switching-key operation has been carried out by the user and that transition to the active mode has taken place from either the inactive mode or sleep mode. Event parameters are not used with this event.

- `CLOCK_TICK_EVENT`

This event indicates that the native clock's time has changed. In DoJa-2.0 Profile and DoJa-3.0 Profile, support is provided for the generation of events in minute units — in other words, this event is generated whenever the time on the native clock reaches a new minute position (i.e., 00 seconds). In order to facilitate the generation of this event, it is necessary to call the `MApplication.setClockTick()` method so that the event may be activated.

This event fires no matter which state the stand-by application is in, but events fired during periods in which the stand-by application is interrupted are suppressed.

The number of seconds which have elapsed from 0 hours, 0 minutes, and 0 seconds on the current day is passed to this event's parameters.

- `WAKEUP_TIMER_EVENT`

This event indicates expiration of the wake-up timer (i.e., the timer which provides notification using a system event that the time set by the stand-by application has expired). The wake-up timer will continue to operate even if the stand-by application is suspended or restarted; however, if the timer expires while the application is suspended, the event notification will be delayed until execution of the stand-by application is restarted.

Operation of the wake-up timer is started by calling the `MApplication.setWakeupTimer()` method (Operation occurs only once; does not function as an interval timer). Furthermore, if the `MApplication.resetWakeupTimer()` method is called, it will be possible to stop the wake-up timer which is currently operating. If the `MApplication.getWakeupTimer()` method is called while a wake-up timer is operating, it will be possible to acquire the time from the current time until said timer will expire.

This event will be generated regardless of the mode in which the stand-by application is currently operating.

Upon generation of this event, a value of 0 will be passed to its parameters if the event occurred as scheduled, and a value of 1 will be passed if occurrence of the event was delayed as a result of the stand-by application's execution being suspended.

- `FOLD_CHANGED_EVENT`

Relevant to mobile phones with a fold or flip design, this event indicates when there is a change in the folded condition or the open/closed condition. Note that this event will only be generated in the case of mobile phones with a fold or flip design.

This event occurs no matter which state the stand-by application is in, but events occurred during periods in which the stand-by application is interrupted are suppressed.

Upon generation of this event, a value of 0 will be passed to its parameter if the closed condition was adopted, and a value of 1 will be passed if the open condition was adopted.

9.2.4 ADF Declarations for Stand-By Applications

Stand-by applications are required to use the ADF's `MyConcierge` key to declare that they are in fact stand-by applications (i.e., by setting the value "Yes"). If this key has not been set, it will not be possible to register the corresponding i- α pli as a stand-by application in the mobile phone.

It is not possible to use an ADF set with a new `MyConcierge` key to upgrade a downloaded normal i- α pli on the cellular phone as a stand-by application. Conversely, it is not possible to use an ADF with its `MyConcierge` key deleted to upgrade a downloaded stand-by application on the cellular phone as a normal i- α pli. In both of these cases, it will be necessary to delete the installed i- α pli from the mobile phone and then to perform a fresh download.

9.3 Launching of Other Functions During Execution of Stand-By Applications

In situations where an attempt is made to launch one of the mobile phone's other functions while a stand-by application is executing, it is a fundamental principle that, as long as the stand-by application is not executing in the active mode, priority be given to said other function. The following section will describe the behavior of stand-by applications when an attempt is made during its execution to launch another function.

Function launched	Behavior of stand-by application	
	In the active mode	In the inactive or sleep mode
Launching through user operation		
Browser launching	Not possible (*1)	The stand-by application is shut-down and the browser is launched. When the user operation is completed and the mobile phone returns to stand-by mode, the stand-by application is re-launched.
Mailer launching	Not possible (*1)	The stand-by application is shut-down and the mailer is launched. When the user operation is completed and the mobile phone returns to stand-by mode, the stand-by application is re-launched.
Voice-call	Not possible (*1)	The stand-by application is suspended and the phone switches to voice-call operations. When the call is completed, the application is restarted and the inactive mode is adopted.
i-appli launching	Not possible (*1)	The stand-by application is shut-down and the i-appli is launched. When the user operation is completed and the mobile phone returns to stand-by mode, the stand-by application is re-launched.
Launching from the stand-by application through application linking		
Browser launching	Not possible (*2)	Not possible (*2)
Voice-call	The stand-by application is suspended and the phone switches to voice-call operations. When the call is completed, the stand-by application is restarted and the active mode is adopted.	Not possible (*3)
i-appli launching	The stand-by application is ended and the specified i-appli launches. When the user operation finishes and the stand-by mode is restored, the stand-by application is restarted.	Not possible. (*3)
i-appli update functionality launching	The stand-by application is ended and JAM (i-appli update functionality) is launched. When the user operation finishes and the stand-by mode is restored, the stand-by application is restarted. (*4)	Not possible. (*3)
Native feature requiring user operation is called	The stand-by application is suspended and the native feature is called. When the user operation finishes and the native feature ends, the stand-by mode is restored and the stand-by application is restarted in the active mode.	Not possible. (*3)
Other-function launching other than the above		
Voice-call	The stand-by application is suspended and the phone switches to voice-call operations. When the call is completed, the application is restarted and active mode is adopted.	The stand-by application is suspended and the phone switches to voice-call operations. When the call is completed, the stand-by application is restarted and the inactive mode is adopted.
Mail or message receipt	Reception of mail or messages is not carried out and execution of the stand-by application is continued. The mobile phone displays a symbol informing the user that non-received mail is being stored on the i-mode server.	The stand-by application is suspended and the phone switches to mail or message receipt operations. When reception is completed, the application is restarted and the inactive mode is adopted. However, if mail or message reception is carried out while an application in the inactive mode is carrying out HTTP communication processing, actions for the active mode will be implemented.
i-appli timer launching	Launching of the i-appli timer is not carried out and execution of the stand-by application is continued. It will be possible for the user to carry out menu operations or the like to find out that timer launching was not possible for an i-appli.	The stand-by application is shut-down and the i-appli timer is launched. When the i-appli which launched by the timer ends and the mobile phone returns to stand-by mode, the stand-by application is re-launched.
Automatic launching of a scheduled native application, such as the alarm, or scheduler.	The stand-by application is suspended and the native application is called. When the native application ends, the stand-by application is restarted in the active mode.	The stand-by application is suspended and the native application is called. When the native application ends, the stand-by application is restarted in the inactive mode.

(*1) Since key events are notified to the stand-by application when executing in active mode, it is not possible for the user to carry out these operations.

(*2) It is not possible to use the application link function (i.e., the `MApplication.launch()` method) from a stand-by application in order to launch a browser.

(*3) Inactivated state stand-by applications cannot launch these features.

(*4) When an i-appli has finished updating with the i-appli update feature, that i-appli will be automatically launched. Stand-by applications will be launched as normal launch. A stand-by application will be launched in stand-by mode after it is launched normally, ends, and return to stand-by mode.

9.4 Precautions for i-appli API Usage

In situations where a stand-by application is launched from stand-by mode, the resulting characteristics lead to a different set of precautions for i-appli API usage than are relevant for normal i-applis. This section will provide a description of these precautions.

9.4.1 User Interface

- The soft-key labels set for frames will not be displayed while a stand-by application is in the inactive mode or sleep mode. Specifically, the native OS uses the display area for soft-key labels when in either of these modes. When the stand-by application switches to the active mode, the label display range corresponding to the soft-key labels as set for the current frame will be displayed.
- When using a `Panel`, it will not be possible to carry out operations relevant to focus (i.e., setting of the focus manager or switching of focus between components) from a stand-by application in the inactive mode. Even if a method for focus operation is called by a stand-by application in the inactive mode, that method will be returned to the application without having done anything.
- When using a `Canvas`, it will not be possible to acquire the keypad condition (i.e., using the `Canvas.getKeyPadState()` method) when in the inactive mode. If a stand-by application calls this method while in the inactive mode, the value 0 will be returned, indicating that no key is being pressed.
- The following tables indicate whether or not notification is carried out for stand-by applications in the inactive mode in terms of both high-level events and low-level events. As a fundamental principle, events associated with key operations are not notified to stand-by applications executing in the inactive mode.

High-level events

Event type	Notification	Remarks
Component event		
BUTTON_PRESSED	×	
SELECTION_CHANGED	○	Notification is provided when the <code>select()</code> or <code>deselect()</code> methods from the <code>ListBox</code> class are used to change the selection condition.
TEXT_CHANGED	○	Notification is provided when the <code>setText()</code> method from the <code>TextBox</code> class is used to perform the resetting of text.
Key event	×	
Soft-key event	×	
Media event	○	
Timer event	○	

Low-level events

Event type	Notification	Remarks
Key-press event	×	Including soft-key operations.
Key-release event	×	Including soft-key operations.
Resume event	○	
Update event	○	
Timer event	○	

- Although scroll operations can be carried out for `Ticker` in the active and inactive modes, this will not be possible upon the adoption of the sleep mode. Scroll operations will become possible once again upon transition from the sleep mode to either active mode or inactive mode.
- Although presenter objects (i.e., `VisualPresenter` and `AudioPresenter`) can perform playback processing when in the active mode or inactive mode, any presenter objects currently being played back will be stopped when the stand-by application adopts inactive mode. No playback-stop event will be generated at that time; furthermore, even if said stand-by application returns from inactive mode to either active mode or inactive mode, the playback will not start again automatically.

- A timer (i.e., `Timer` or `ShortTimer`) will be able to operate when a stand-by application is in either active mode or inactive mode. If a stand-by application which is operating a timer switches to the sleep mode, said timer will be stopped (i.e., as if the `stop()` method had been called). Even if the stand-by application subsequently returns from the sleep mode to either active mode or inactive mode, operation of the timer will not start again automatically.
- Inactive mode stand-by applications cannot launch the IME from `Canvas` (call the `Canvas.imeOn()` method). If inactive mode stand-by applications attempts to launch the IME, an exception will occur.

9.4.2 Input and Output

- When a stand-by application in the inactive mode is carrying out HTTP(S) communication, this communication will be suspended not only when a voice call is received, but also when a voice call is initiated as a result of user operations. As with the suspension of communication due to an incoming voice call, it will be necessary to retry the communication processing in such a case. For more details on the conditions for the suspension and restart of i-applis, refer to Section 3.6.
- Stand-by applications are capable of HTTP(S) communication only when in the active mode or inactive mode. If while a thread is carrying out HTTP(S) communication, another thread calls the `MApplication.sleep()` method to implement the sleep mode, said communication will be suspended and an exception will be generated. When the stand-by application subsequently returns to one of the other modes, the thread which had been carrying out communication will start to operate again from exception processing.
- If while the ScratchPad is writing data to non-volatile memory, a thread calls the `MApplication.sleep()` method to implement the sleep mode, transition to this mode will be delayed until the writing of said data has been completed. Note that if the sleep mode is adopted while the ScratchPad is open, this open condition will be maintained even upon transition from sleep mode to either active mode or inactive mode, and it will be possible to continue with the writing or reading of data.
- Communication with external devices through the use of OBEX external connection will be possible only when the stand-by application is in the active mode. If a stand-by application using OBEX external connection to perform communication switches to inactive mode, said communication will be suspended and an exception will be generated. Note that this applies both to cases where the stand-by application is operating as an OBEX client and where it is operating as an OBEX server.

9.4.3 Hardware Control

- It is not possible for stand-by applications in inactive mode to carry out back-light control. Even if a stand-by application in this mode calls the `PhoneSystem.setAttribute()` method to specify modification of the back-light condition, that method will be returned to the application without having done anything.
- Although it is possible for stand-by applications in the active mode to carry out back-light control, a back-light which has been turned on by an application will be automatically turned off by the system after a specific period of time has elapsed.
- Control of the vibrator can be carried out by stand-by applications both in the active and in inactive modes. If a stand-by application switches to the sleep mode while the vibrator is currently turned on, the vibrator will be stopped. Furthermore, even if that stand-by application returns to either the active mode or in inactive mode, the vibration will not be automatically restarted.
- In the case of all other hardware resources for which the user can perform menu operations to make settings, as a fundamental principle, priority is given to user settings when an attempt is made to change modes from a stand-by application.

9.4.4 Application Linking

- When a stand-by application has been launched in stand-by, it cannot use the `MApplication.launch()` method to launch the browser under any circumstance. If a stand-by application attempts to launch a browser, an exception will be generated.
- Inactive mode stand-by applications cannot call application linking methods which will require a user operation (such as confirmation of permission to place an outgoing voice-call with the `Phone.call()` method). If an inactive mode stand-by applications attempts to call these methods, an exception will occur.

Chapter 10

OBEX External Connection

OBEX external-connection functionality enables i-appli to connect to, and communicate with, external devices via the infra-red port, and this functionality is supported as i-appli standard API with DoJa-2.0 Profile or later. In this, the term “external device” also includes mobile phones which are compatible with DoJa-2.0 Profile or later; furthermore, support is provided in i-appli specification for both client API and server API in an OBEX external-connection setup.

Accordingly, i-applis for the exchange of business cards, schedules, and other small data items between mobile phones may be easily developed.

This chapter will provide a description of the important points that should be known when developing systems intended for OBEX client API, OBEX server API, and external devices other than mobile phones within the i-appli Runtime Environment.

Note that although OBEX external-connection functionality is implemented in accordance with IrDA standards, said standards will not be dealt with directly in this manual. For more details regarding the IrDA standards, refer to the relevant documentation as published by the Infrared Data Association. Furthermore, OBEX external-connection functionality within DoJa-2.0 Profile and DoJa-3.0 Profile conforms with the following versions of the IrDA standards.

- Infrared Data Association “Link Manager Protocol version 1.1”
- Infrared Data Association “Serial Infrared Physical Layer Specification version 1.3”
- Infrared Data Association “Tiny TP: A Flow-Control Mechanism for use with IrLMP version 1.1”
- Infrared Data Association “Object Exchange Protocol IrOBEX version 1.2”
- Infrared Data Association “Specification for Ir Mobile Communications(IrMC) version 1.1”
- Infrared Data Association “Serial Infrared Link Access Protocol version 1.1”

10.1 OBEX Data Communication

OBEX (IrOBEX) standardizes the procedures for data communication using the infra-red port in a format known as “object exchange.” The term “object” as used here refers to a grouping of data, such as commonly manifested in a file.

Although OBEX standardizes the sequence (protocol) for the exchange of data, it does not aim to standardize API. i- α pli specification sets forth regulations for API in order to facilitate the handling of OBEX data communication on a Generic Connection framework.

HTTP was referenced in the determination of the OBEX specification, and this specification conforms with the client-server model wherein the OBEX client sends a request and the OBEX server returns the corresponding response. In particular, the OBEX client requests that the OBEX server carries out processing of an operation (which corresponds to a method in terms of HTTP), and the OBEX server then returns either a response status as the result of the operation or an object’s contents as the results of processing. Within the i- α pli Runtime Environment are provided with the following two operations in order to facilitate the exchange of objects between clients and servers.

- GET operation: The OBEX client requests the OBEX server to transfer an object.
- PUT operation: The OBEX server receives an object from the OBEX client.

With the exception of the fact that it is represented in binary format, this protocol resembles HTTP particularly in terms of the stipulated response-status content and also in that both requests and responses comprise a header and body.

The OBEX standard sets forth detailed procedures for the division and reconstruction of objects which are difficult to handle as a result of, for example, the packet-length restrictions applicable to infra-red communication. However, since the API provided by the i- α pli Runtime Environment conceal these complicated regulations, as a developer, you will be able to create this type of i- α pli with relative ease.

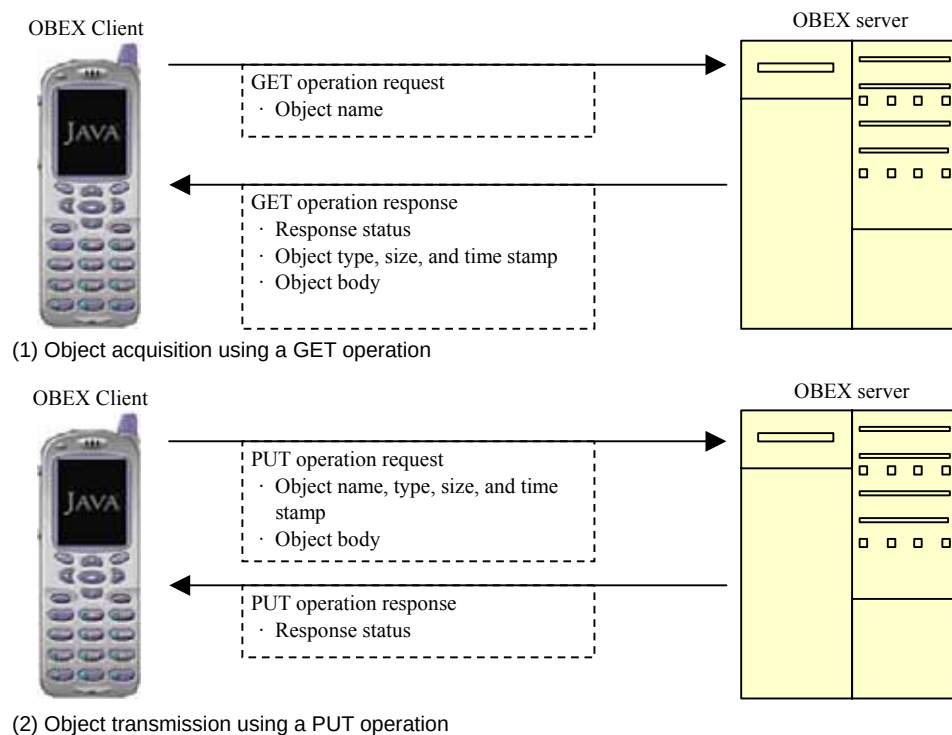


Figure 11: Typical OBEX Client and Server Implementation

10.2 OBEX External-Connection API

The API for OBEX external connection are provided as a unique i-appli API set, constructed on the Generic Connection framework. The makeup of the OBEX external-connection API is shown in the following diagram.

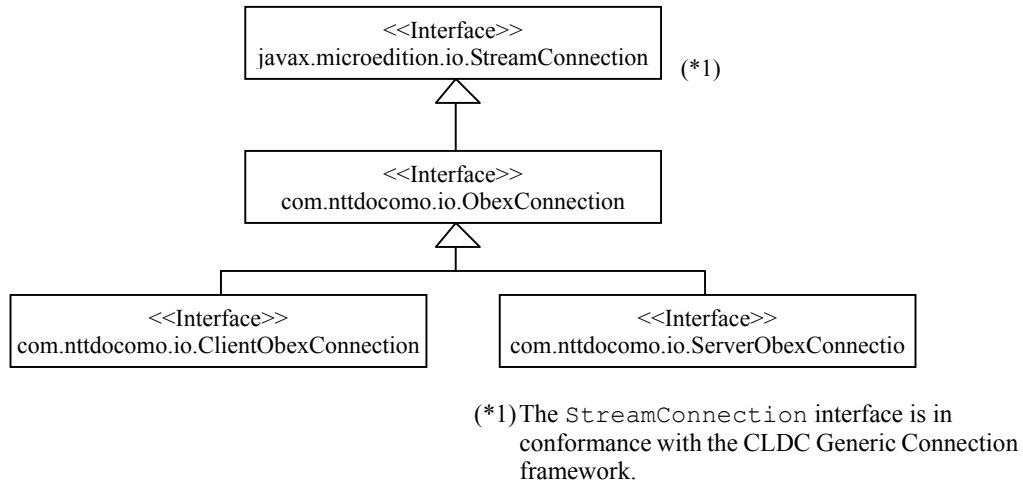


Figure 12: Makeup of OBEX External-Connection API

The `ObexConnection` interface defines the constants and access methods as used commonly by the OBEX client and the OBEX server; furthermore, `ClientObexConnection` and `ServerObexConnection` are provided as sub-interfaces inherited from the `ObexConnection` interface. The specific methods for OBEX clients and OBEX servers are defined using these two sub-interfaces.

Although the connection objects used for OBEX external connection are acquired through the use of the same `Connector.open()` method as for HTTP communication and the ScratchPad, this connection implements either `ClientObexConnection` or `ServerObexConnection`. The URL specified for the `Connector.open()` method is different for OBEX client applications and OBEX server applications.

The following sections will provide individual descriptions of OBEX client API and OBEX server API.

10.2.1 OBEX Client API

The connection object used by the OBEX client implements the `ClientObexConnection` interface. In order to acquire this connection object, application programs must specify the URL string "`obex:/irclient`" and call the method `Connector.open()`.

The following presents an example of acquisition and usage of the `ClientObexConnection` implementation interface in an application program.

Example: Usage of ClientObexConnection

```

ClientObexConnection coc;
int response;
try {
    // Acquire the ClientObexConnection instance and
    // connect to the OBEX server.
    coc = (ClientObexConnection)Connector.open("obex:/irclient",
                                                Connector.READ_WRITE,
                                                true);

    coc.connect();

    // Execute the GET operation.
    coc.setOperation(ObexConnection.GET);           // Set GET for operation.
    coc.setName("userdata_1");                      // Set the name of the object for GET.
    coc.sendRequest();                              // Send the operation.

    // Check the response code returned from the OBEX server and receive
    // if normal. Read the object body.
    response = coc.getResponseCode();
    if (response==ObexConnection.SUCCESS) {
        InputStream in = coc.openInputStream();
        // Read and process the data requested with the GET operation.
        :
        in.close();
    } else {
        throw new IOException();
    }

    // Execute the PUT operation
    coc.setOperation(ObexConnection.PUT);           // Execute the PUT operation.
    coc.setName("userdata_2");                      // Set PUT for operation.
    coc.setType("text/plain");                      // Set the name of the object for
                                                    // PUT.
    coc.setTime(System.currentTimeMillis());        // Set the time stamp of the object
                                                    // for PUT.
    OutputStream out = coc.openOutputStream();
    // Write and process the data to be sent with the PUT operation.
    :
    out.close();
    coc.sendRequest();                              // Send the operation.
    // If the response code is normal, close ClientObexConnection and end
    // processing.
    response = coc.getResponseCode();
    if (response!=ObexConnection.SUCCESS) {
        throw new IOException();
    }
    coc.close();
} catch (IOException ioe) {
    // Exception processing.
    // Processing for IOException and ConnectionException as thrown by API,
    // and for IOException as thrown by the application program.
    :
}

```

When an application program is using ClientObexConnection, it is first of all necessary to call the connect() method to establish an infra-red communication link with the OBEX server. Once a communication link has been established, it will be possible to process multiple operations in sequence. In the above sample code, a GET operation and a PUT operation are each processed once on the established one-off communication link, and after that, the link is severed using ClientObexConnection.close().

The processing of operations on the OBEX client is carried out in accordance with the following sequence.

1. The `setOperation()` method is used to set the operation type.
2. The access method defined by `ObexConnection` is used and the setting of OBEX request data is carried out. A Name header, Type header, and Time header are contained within the OBEX request headers which may be set using the access method.
3. In the case of PUT operations, the content of the object body to be sent is written. This operation is carried out using the output stream acquired using the connection object's `openOutputStream()` or `openDataOutputStream()` method.
4. The `sendRequest()` method is used to send an operation-processing request to the OBEX server. This method is blocked until a response is returned from the OBEX server. When returning after the calling of this method, the content of the OBEX response code or OBEX response header as returned from the OBEX server may be referenced if so required. A Name header, Type header, and Time header are contained within the OBEX response headers which may be referenced by the OBEX client; furthermore, the OBEX response code is defined by `ObexConnection`. The OBEX response code as defined by `ObexConnection` conforms with the response code regulations from the OBEX standards. For more details regarding the detailed meaning of each code, refer to the OBEX standards. In situations where the OBEX server can perform normal processing for the OBEX client's request, it is standard for the OBEX client application to receive `ObexConnection.SUCCESS` from the i-appli Runtime Environment (i.e., `ClientObexConnection.getResponseCode()` method) as an OBEX response code.
5. In the case of GET operations, the content of the returned object body is read. This operation is carried out using the input stream acquired using the connection object's `openInputStream()` or `openDataInputStream()` methods. It is possible to acquire the length of the object body using the connection object's `getContentLength()` method.

Please ensure that processing on the OBEX client conforms with the above sequence as a whole. For example, if an attempt is made to acquire the OBEX response header in a situation where no operation processing request is issued to the OBEX server, an exception will be generated.

The following diagram illustrates condition transitions for `ClientObexConnection`.

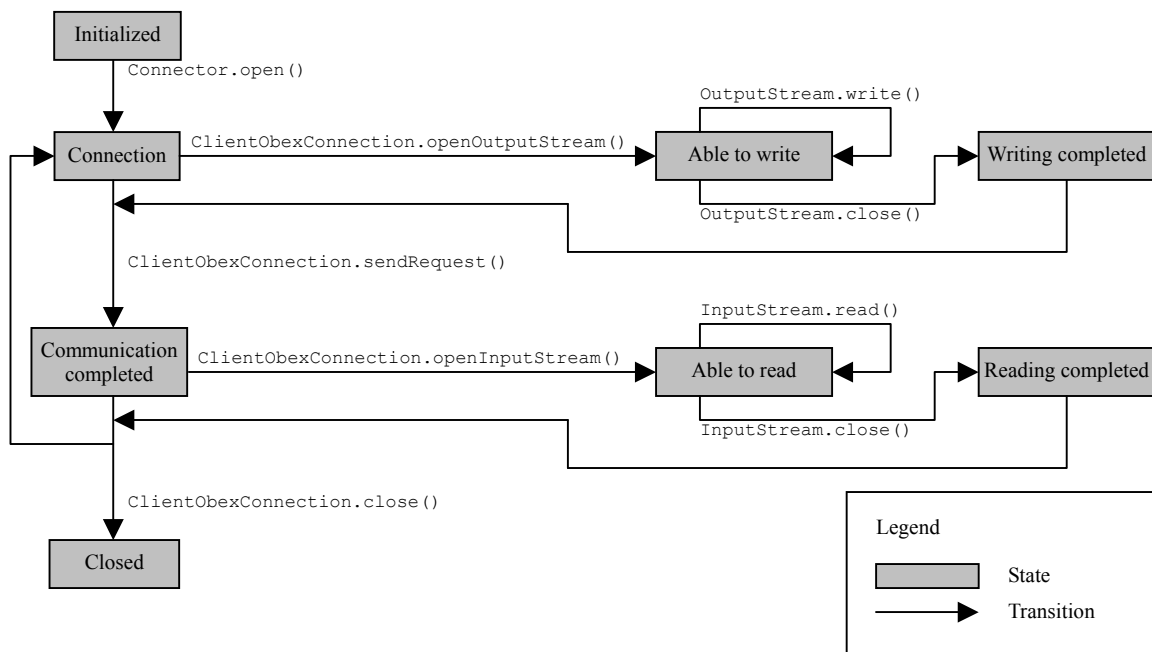


Figure 13: Condition Transitions for `ClientObexConnection`

Notes:

- With OBEX external connection, it is not possible to retrieve an I/O stream using the methods `openInputStream()`, `openDataInputStream()`, `openOutputStream()`, or `openDataOutputStream()` directly from the `Connector` class. In order to do this, the application program must first of all acquire an `ClientObexConnection` object using the `Connector.open()` method, and the I/O stream can be retrieved from this object.
- Communication using the infra-red port of a mobile phone is carried out automatically by the system when `ClientObexConnection.connect()` is called from the application program.
- It is not possible to use a single operation for an OBEX server to carry out both the transmission of an object and the receipt of objects from the server. OBEX clients may only carry out the transmission of objects using the PUT operation, and the receipt of objects using the GET operation.
- In terms of the open mode for connection objects on OBEX clients, the READ mode corresponds to GET operations, and the WRITE mode corresponds to PUT operations. If it is intended to use both GET and PUT operations during a single connection, please ensure that READ_WRITE mode is used.
- Values for Name headers and Type headers can be specified up to 63 characters. The Type header value can use ASCII characters only.
- The number of communication links which can be established at any one time from a single i-appli is one only.
- In a situation where a thread has called either the `connect()` or `sendRequest()` method and is waiting for an action from the OBEX server, if another thread calls the `ClientObexConnection` object's `close()` method, the connection will be severed. Furthermore, it is also possible that the connection may be severed as a result of factors such as time-out occurrence or severing operations by the user.
- For both the GET and PUT operations, the handling of objects up to 100 Kbytes in size is possible. It is not possible to carry out the transmission or reception of objects in excess of this limit.

[DoJa-3.0]

Size restrictions for objects that can be sent/received have been relaxed in DoJa-3.0 Profile. In the DoJa-2.0 Profile, the size of objects that can be sent/received is limited to 30KB.

- The operation type, OBEX header, object-data content, and other similar items as set by the application program are valid for a single operation only. In situations where operations are to be continued for a single connection, please ensure that these settings are redone for each operation.
- If the response code returned from an OBEX server contains a value indicating an error (i.e., a code not of the order 0x20), then the i-appli Runtime Environment will throw an exception.
- Mobile phones which are compatible with DoJa-2.0 Profile or later include a feature known as Self Mode which stops all external connection and allows usage only of functions that do not perform communication. Although it will be possible for i-applis to launch when the mobile phone has been set to Self Mode by the user, it will not be possible for these i-applis to carry out OBEX external connection.
- Stand-by applications in the inactive mode cannot perform OBEX external-connections. When a stand-by application in the active mode is performing an OBEX external-connection, the OBEX external-connection will fail if the application attempts to transition into the inactive mode.
- When i-applis are carrying out communication using the infra-red port, the i-appli Runtime Environment will display a message which informs of this situation. Similarly, in situations where a communication link cannot be established or where communication is discontinued as a result of user operations or the like, the i-appli Runtime Environment will display a message which informs of the corresponding situation. It is not possible to prevent the display of these messages.

10.2.2 OBEX Server API

The connection object used by the OBEX server implements the `ServerObexConnection` interface. In order to acquire this connection object, application programs must specify the URL string "obex://irserver" and call the method `Connector.open()`.

The following presents an example of acquisition and usage of the `ServerObexConnection` implementation interface in an application program.

Example: Usage of ServerObexConnection

```

ServerObexConnection soc;
try {
    // Acquire the ServerObexConnection instance
    soc = (ServerObexConnection)Connector.open("obex://irserver",
                                                Connector.READ_WRITE,
                                                true);

    // Await a request for establishment of a communication link from the OBEX client.
    soc.accept();

    // Process the operation request from the OBEX client.
    while (true) {

        soc.receiveRequest(); // Await an operation request.
        int operation = soc.getOperation(); // Acquire the type for the arriving
                                           // operation.

        if (operation==ObexConnection.GET) { // GET operation request
            // Acquire the name of the requested object.
            String name = soc.getName();
            // Set the header data for the object corresponding to the
            // specified name.
            soc.setType("text/plain");
            soc.setTime(System.currentTimeMillis());
            // Carry out write processing for the object corresponding to the
            // specified name.
            OutputStream out = soc.openOutputStream();
            :
            out.close();
            // Return a status code indicating normal completion to the OBEX
            // client.
            soc.sendResponse(ObexConnection.SUCCESS);

        } else if (operation==ObexConnection.PUT) { // PUT operation request
            // Acquire the name data for the transmitted object.
            String name = soc.getName();
            String type = soc.getType();
            long time = soc.getTime();
            // Carry out read processing for the transmitted object.
            InputStream in = soc.openInputStream();
            :
            in.close();
            // Return a status code indicating normal completion to the OBEX
            // client.
            soc.sendResponse(ObexConnection.SUCCESS);

        } else if (operation==ObexConnection.DISCONNECT) { // Disconnection
                                                            // request
            break;
        }
    }

    // Sever the communication link and end communication.
    soc.close();
} catch (IOException ioe) {
    // Exception processing.
    // Processing for IOException and ConnectionException as thrown by API.
    :
}

```

When an application program is using `ServerObexConnection`, it is first of all necessary to call the `accept()` method to await a link establishment request for infra-red communication link from the OBEX client (i.e., execution of the OBEX client's `connect()` method). Once a communication link has been established, it will be possible to process multiple operations in sequence. In the above sample code, the processing of operations is carried out repeatedly on a single established link until a disconnection request is received from the OBEX client.

The processing of operations on the OBEX server is carried out in accordance with the following sequence.

1. The `receiveRequest()` method is used to receive requests. This method is blocked until the receipt of a request from the OBEX client.
2. The `getOperation()` method is used to acquire the request operation's type. If `ObexConnection.DISCONNECT` is received as this type, it indicates notification that the OBEX client is disconnecting the communication link. In such a case, please ensure that the OBEX server also carries out connection-object closure.
3. The content of the OBEX request header is acquired using the access method as defined by `ObexConnection`. A Name header, Type header, and Time header are contained within the OBEX request headers which may be set using the access method.
4. In the case of a PUT operation, the content of the body for the object sent from the OBEX client is read. This operation is carried out using the input stream retrieved using the connection object's `openInputStream()` or `openDataInputStream()` method. The length of the object body can be acquired using the connection object's `getContentLength()` method.
5. The access method defined by `ObexConnection` is used to set the content of the OBEX response header. A Name header, Type header, and Time header are contained within the OBEX response headers which may be set using the access method.
6. In the case of a GET operation, the body for the object requested by the OBEX client is written. This operation is carried out using the output stream retrieved using the connection object's `openOutputStream()` or `openDataOutputStream()` method.
7. A response is sent to the OBEX client using the `sendResponse()` method. This method's parameters are specified using the OBEX response code as defined by `ObexConnection`. Note that the OBEX response codes defined by `ObexConnection` conform with the response code rules from the OBEX specifications. For more details regarding the meaning of the various codes, refer to the OBEX specifications. Generally speaking, when an OBEX server is correctly able to process the request from the OBEX client, the OBEX server application will return `ObexConnection.SUCCESS` to the i-appli Runtime Environment (`ServerObexConnection.sendResponse()` method) as an OBEX response code.

Please ensure that processing on the OBEX server conforms with the above sequence as a whole. For example, if an attempt is made to acquire the OBEX request header in a situation where operation processing requests are not being received, an exception will be generated.

The following diagram illustrates condition transitions for `ServerObexConnection`.

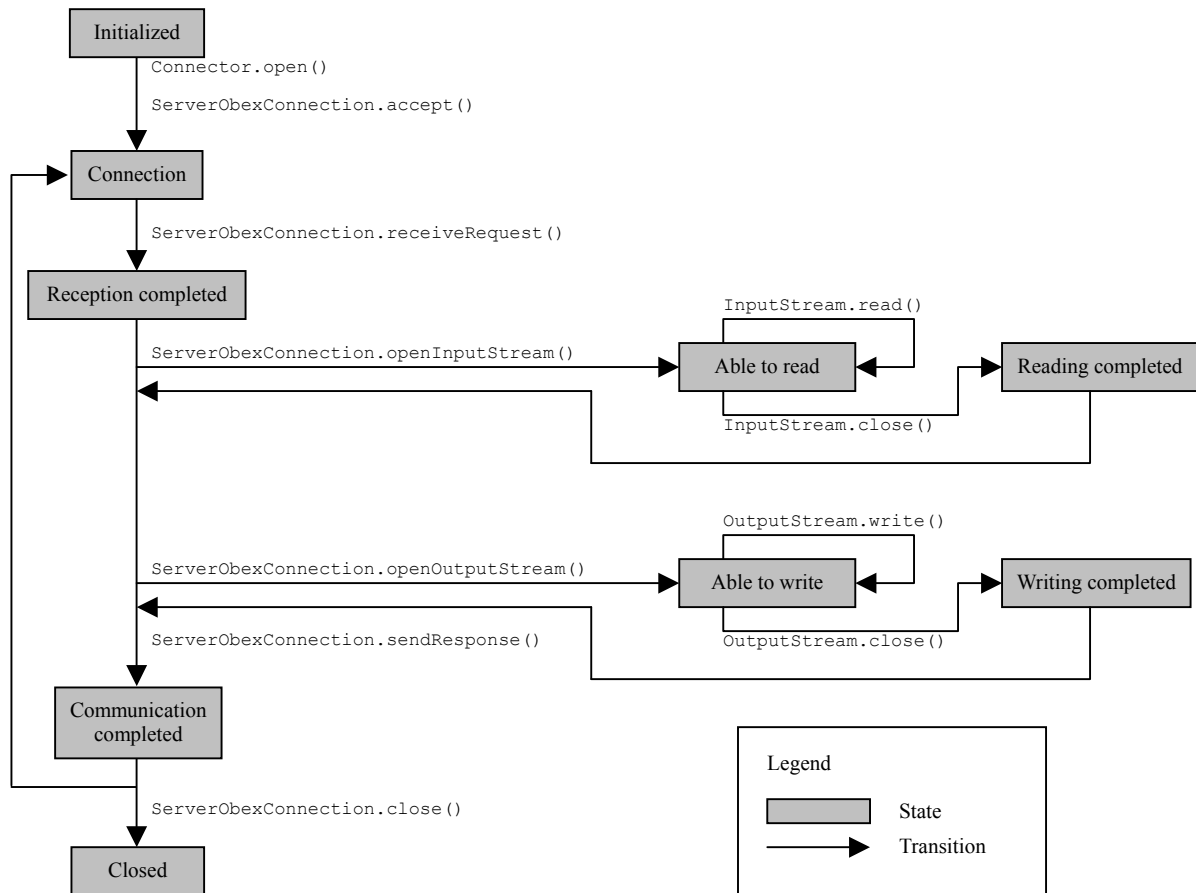


Figure 14: Condition Transitions for ServerObexConnection

Notes:

- With OBEX external connection, it is not possible to retrieve an I/O stream using the methods `openInputStream()`, `openDataInputStream()`, `openOutputStream()`, or `openDataOutputStream()` directly from the `Connector` class. In order to do this, the application program must first of all acquire an `ClientObexConnection` object using the `Connector.open()` method, and the I/O stream can be retrieved from this object.
- Communication using the infra-red port of a mobile phone is carried out automatically by the system when `ServerObexConnection.accept()` is called from the application program.
- It is not possible for an OBEX server to carry out the transmission of an object in response to a PUT operation from the OBEX client. The OBEX server can only carry out object transmission in response to a GET operation.
- In terms of the open mode for connection objects on OBEX servers, the READ mode corresponds to PUT operations, and the WRITE mode corresponds to GET operations. If it is intended to use both GET and PUT operations during a single connection, please ensure that READ_WRITE mode is used.
- Values for Name headers and Type headers can be specified up to 63 characters, respectively. The Type header value can use ASCII characters only.
- The number of communication links which can be established at any one time from a single i-appli is one only.
- In a situation where a thread has called either the `accept()` or `receiveRequest()` method and is waiting for an action from the OBEX client, if another thread calls the `ServerObexConnection` object's `close()` method, the connection will be severed. Furthermore, it is also possible that the connection may be severed as a result of factors such as time-out occurrence or severing operations by the user.
- For both the GET and PUT operations, the communication of objects up to 100 Kbytes in size is possible. It is not possible to carry out the transmission or reception of objects in excess of this limit.

[DoJa-3.0]

Size restrictions for objects that can be sent/received have been relaxed in DoJa-3.0 Profile. In DoJa-2.0 Profile, the size of objects that can be sent/received is limited to 30KB.

- The OBEX header, object-data content, and other similar items as set by the application program are valid for a single operation only. In situations where operations are to be continued for a single connection, please ensure that these settings are redone for each operation.
- Mobile phones which are compatible with DoJa-2.0 Profile or later include a feature known as Self Mode which stops all external connection and allows usage only of functions that do not perform communication. Although it will be possible for i-applis to launch when the mobile phone has been set to Self Mode by the user, it will not be possible for these i-applis to carry out OBEX external connection.
- Stand-by applications in the inactive mode cannot perform OBEX external-connections. When a stand-by application in the active mode is performing an OBEX external-connection, the OBEX external-connection will fail if the application attempts to transition into the inactive mode.
- When i-applis are carrying out communication using the infra-red port, the i-appli Runtime Environment will display a message which informs of this situation. Similarly, in situations where a communication link cannot be established or where communication is discontinued as a result of user operations or the like, the i-appli Runtime Environment will display a message which informs of the corresponding situation. It is not possible to prevent the display of these messages.

10.3 Hints for External Device Connection

The following section will describe the precautions to be observed regarding i-appli's lower-layer specifications for infra-red communication, and also regarding infra-red communication between i-applis on mobile phones and devices other than mobile phones.

It will not be necessary to observe these precautions if you are developing i-applis which will carry out infra-red communication between mobile phones. However, in cases where a system features infra-red communication between an i-appli on a mobile phone and some other external device, please ensure that these precautions are taken into consideration when designing the external device.

Note that more details regarding the following content are available in the IrDA specifications.

<Physical conditions>

Communication speed: The maximum communication speed with common support is 115.2 Kbps.

Communication distance: The infra-red devices fitted into mobile phones conform with the IrMC Low Power Option from IrDA specifications; accordingly, the communication distance is limited to the level of 20 cm in an indoor environment.

<IrLAP connection parameters>

Baud rate: 9,600, and 38,400 bps, and 115,200 bps are supported.

Data size: 64-byte, 128-byte, and 256-byte are supported.

Link disconnect/ threshold time: 3 seconds (threshold = 0 seconds) and 8 seconds (threshold = 3 seconds) are supported.

<IrLMP device names and nicknames>

The following types of character strings are set for device names and nicknames.

DoCoMo/<device-name>/j (Example: DoCoMo/X505i/j)

A name identical to that included in the user agent is set for <device-name>. By referring to this name, the external device can identify the mobile phone model with which it is currently communicating.

<IrLMP Service Hints>

The below flags are set to on in the service hints bit-map.

```

Byte1 : Bit1   PDA/Palmtop
Byte1 : Bit7   Extension
Byte2 : Bit8   Telephony
Byte2 : Bit13  IrOBEX

```

[DoJa-3.0]

In DoJa-3.0 Profile, a 115,200bps Baud Rate has been added to the IrLAP connection parameters. Note that the Baud Rates supported in common with DoJa-2.0 Profile are 9,600bps, and 38,400bps; connections at 115,200bps are not guaranteed.

The details of IrLMP service hint settings in DoJa-2.0 Profile may vary depending on the manufacturer.

Notes:

- Of the operations set by OBEX, the i-appli Runtime Environment does not support SetPath operations for either the OBEX client or OBEX server. Furthermore, the ABORT operation is not supported for the OBEX client.
- OBEX recognition is not supported within the i-appli Runtime Environment.
- Five different types of headers may be received by the OBEX client and the OBEX server within the i-appli Runtime Environment, and these are Name, Type, Time, Body, and End of Body. Furthermore, please ensure that the object body (i.e., Body header and End of Body header) is sent after the other headers.
- The infra-red communication functionality of mobile phones has been certified for mutual IrDA connection. However, this does not necessarily guarantee that connection will be possible with all external devices conforming to the IrDA specifications. Cases may exist where communication is not possible as a result of factors such as the environment used on the external device, the device hardware, or the corresponding software characteristics.

Chapter 11

Application Linking

In DoJa-2.0 Profile, linked operation was supported between i-appli and native mobile phone applications, such as browsers and mailers. In DoJa-3.0 Profile, the scope of linked operation was enlarged (e.g. linking between i-applis) and in addition, support was added for enhanced application linking, such as calls from i-appli to native functionality, like bookmark and image management features, etc. By using this functionality, i-appli can offer services which fully use the functionality of the mobile phone, not just i-appli functionality.

Further, regarding the nature of this functionality, part of the application linking functionality falls within the scope of the trusted i-appli. This book will not cover functionality contained in the trusted i-appli. Rather, the functionality that will be explained is that which is outside the trusted i-appli and can be used by general i-appli too.

The application linking functionality supported by DoJa-2.0 and DoJa-3.0 Profiles is as follows.

Application linked operation

Summary of linked operation	DoJa-2.0	DoJa-3.0
Launching i-appli from browser	Yes	Yes
Launching browser from i-appli	Yes	Yes
Launching i-appli from mailer	Yes	Yes
Launching i-appli from external device (infrared port)	Yes	Yes
Launching i-appli from i-appli	No	Yes
Launching i-appli update functionality from i-appli	No	Yes

Calling native functionality from i-appli

Summary of linked operation	DoJa-2.0	DoJa-3.0
Calling voice call functionality (dialing)	Yes	Yes
Calling voice call functionality (individual identification information referencing)	See (*1)	Yes
Calling phonebook management functionality (creation of phonebook entries and groups)	No	Yes
Calling bookmark management functionality (bookmark creation)	No	Yes
Calling schedule management functionality (schedule creation)	No	Yes
Calling image data management functionality (image creation, selective loading, ID designation loading)	No	Yes
Calling camera functionality (shooting images, retrieving images shot)	See (*2)	Yes

(*1) Individual identification information referencing was added to the i-appli standard API in DoJa-2.1 Profile.

(*2) Calls to camera functionality was partially supported by the i-appli optional API in DoJa-2.0 Profile. In DoJa-3.0 Profile, some of this functionality was enhanced and adopted into the i-appli standard API.

This chapter will provide a description of the usage methods for this application linking functionality, in addition to providing an explanation of factors such as the restrictions applying to certain models.

11.1 Browser Linking

Browser linking allows i- α plis to be launched from a browser and vice-versa. Note, however, that the launching of i- α plis from a browser will only be possible if the i- α pli in question has already been installed in the mobile phone.

11.1.1 Browser Launching of i- α plis

As a developer, you will be required to carry out the following two tasks to ensure that i- α plis can be launched from browsers.

- (1) Create an HTML file for launching the i- α pli on web site which may be accessed using the internet, and include an i- α pli launching tag within this file. The i- α pli which actually launches on the mobile phone when the tag is selected will depend on the content of the i- α pli launching tag.
- (2) Use the ADF `LaunchByBrowser` key to declare that the i- α pli is permitted to be launched from the URL for the HTML created in (1) above. It will not be possible to launch an i- α pli using the HTML at an URL which has not been given the corresponding permission using the `LaunchByBrowser` key. Note that the URL as specified by the `LaunchByBrowser` key will be forward-match compared with the URL for the HTML, and if the content specified in the `LaunchByBrowser` key is all contained in the URL for the HTML, then launch permission will be granted for that HTML.

In order to enable launching of an i- α pli from a browser, it will first of all be necessary to display the HTML for i- α pli launching on the browser. The i- α pli launching tag will be represented as a link, and the user can select this link to launch the corresponding i- α pli on the mobile phone.

The following will provide samples of the HTML tags for i- α pli launching.

```
<OBJECT declare id="application.declaration"
data="http://www.nttdocomo.co.jp/java/abc.jam" type="application/x-jam">
<PARAM name="Param1" value="i-mode">
<PARAM name="Param2" value="i- $\alpha$ pli">
</OBJECT>
In order to launch the i- $\alpha$ pli,
<A ista="#application.declaration" href="notapplicable.html">Click here</A>.
```

The tags used for i- α pli launching are as described below.

- A tag and `ista` attribute

The A tag is used to reference the OBJECT tag which corresponds to the i- α pli for launching. When the user selects this A tag, the i- α pli as installed on the mobile phone will be launched. The `ista` attribute from this tag is used to specify the name from the OBJECT tag's `id` attribute. Specifically, the ADF URL from the OBJECT tag as described below determines which of the i- α plis on the mobile phone will actually be launched.

The `ista` attribute was established in DoJa-2.0 Profile, and is not recognized by DoJa-1.0 Profile mobile phones.

Always specify a `href` attribute in A tags that have an `ista` attribute. With regards to A tags and attributes (`ijam` attribute, `ilet` attribute) for use in application downloading, see Chapter 14.

- OBJECT tag

The OBJECT tag is used to specify the ADF corresponding to the i- α pli; furthermore, this tag's `id` attribute indicates the name (unique in HTML) which will be referenced by the A tag's `ista` attribute. The `data` attribute specifies the URL which indicates the ADF location. When an i- α pli has been downloaded onto the mobile phone, the URL for the ADF corresponding to that i- α pli will be recorded. When i- α pli launch from the browser is then utilized, the i- α pli referencing the ADF with the same URL as that specified in the `data` attribute will be launched.

The `type` attribute is used to specify the content type for the data displayed by the `data` attribute (in this case, ADF). The ADF's content type is "application/x-jam"

It is also possible to include the following PARAM tag within the OBJECT tag.

- PARAM tag

The PARAM tag is used upon the launching of an i-appli from a browser to specify the parameters which can be acquired using the `IApplication.getParameter()` method.

Up to a maximum of sixteen PARAM tags may be included within a single OBJECT tag. Furthermore, the total of all of the name attribute lengths and value attribute lengths as specified for all PARAM tags within a single OBJECT tag is limited to a maximum of 255 bytes. In the case of both name and value attributes, the value SJIS can be used to indicate Japanese text.

The following diagram illustrates the relationship between an i-appli downloaded onto a mobile phone and the content of the HTML for launching of that i-appli.

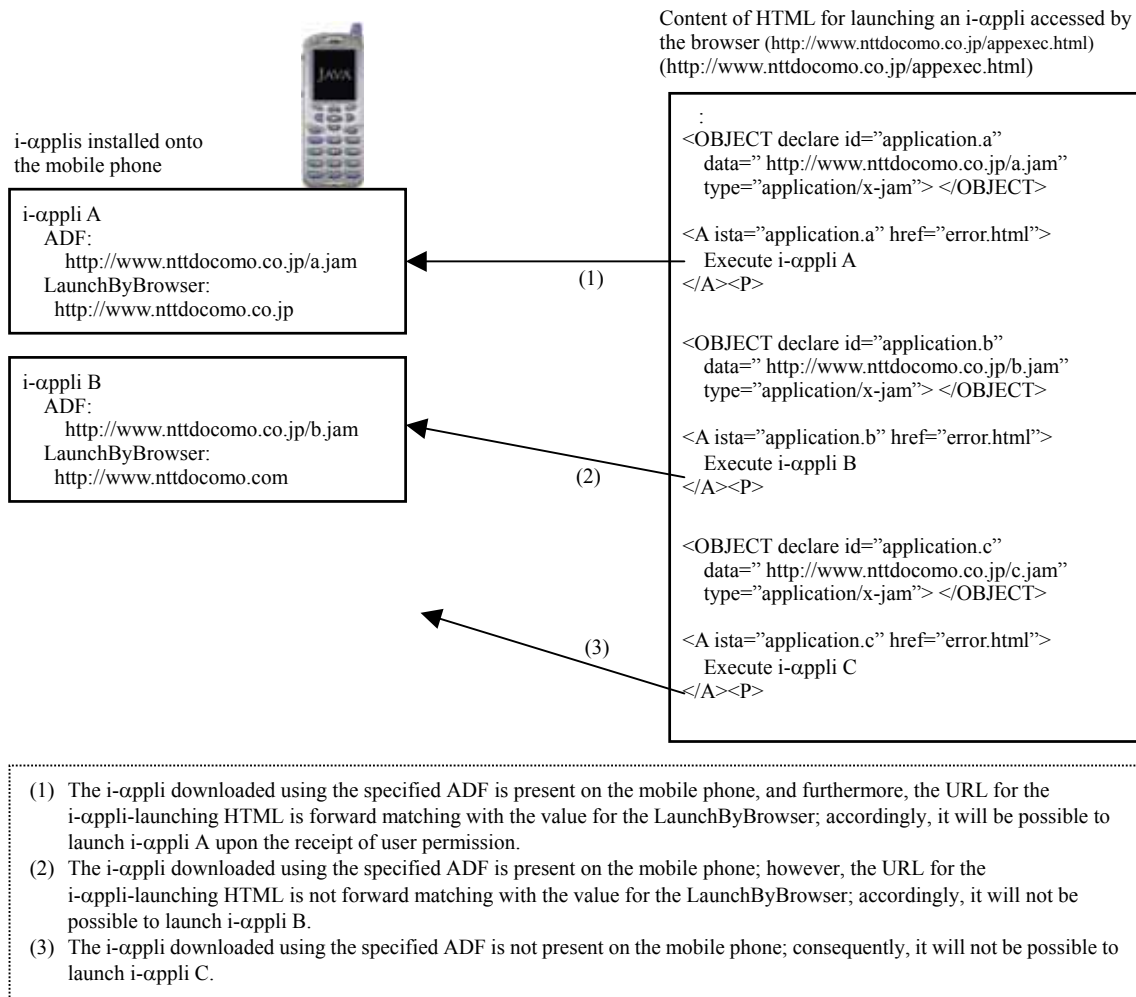


Figure 15: Downloaded i-applis and HTML for i-appli Launching

Notes:

- The user of a mobile phone may use that mobile phone's settings to forbid the launching of i-applis from browsers. Furthermore, before an i-appli is launched from a browser, the i-appli Runtime Environment will request confirmation from the user that i-appli launching is to be permitted. If the permission of the user is not obtained through either of these methods, it will not be possible to launch the corresponding i-appli.
- When the operation of an i-appli as launched from a browser comes to an end (i.e., ended by `IApplication.terminate()`), the mobile phone will return to the condition which existed before launching of the i-appli. However, in the i-appli launched from the browser is forced to end by the user, or the i-appli launched from the browser operates or calls another native application with application linking, the state entered after i-appli ends will depend on the manufacturer.

11.1.2 i-appli Launching of Browsers

The `IApplication.launch()` method is used to launch the browser from i-appli. When launching the browser, this method's first argument (type of application to be launched) will be set to `IApplication.LAUNCH_BROWSER`, while in the first element of the second argument (String array indicating launch arguments) the target URL (http and https schemes are supported) will be set. The URL can be specified up to 255 bytes.

Note that the i-appli launching the browser must use the ADF's `UseBrowser` key to declare the launching of a browser (by setting the value to "launch"). i-applis for which this declaration has not been carried out will be unable to launch browsers.

Notes:

- Stand-by applications launched in stand-by, whether in the active mode or inactive mode, cannot launch the browser. However, the same stand-by application, if launched normally, can launch the browser.
- If the `IApplication.launch()` method is called, the i-appli will adopt the suspended condition. The browser will subsequently be launched if the following two conditions are satisfied.
 - Packet communication is possible (i.e., when the "i" mark as used on the mobile phone screen to indicate the communication condition is either lit or flashing)
 - Before browser launching, i-appli Runtime Environment requests user permission for this operation and it is granted.

If either of these conditions is not satisfied, the i-appli will resume and the `IApplication.launch()` method will be returned to its point of calling without an exception being thrown. Furthermore, when the browser is launched as a result of both conditions being satisfied, the i-appli will be terminated.

- If the i-appli from which the browser was launched is terminated as a result of user operations, the stand-by condition will once again be adopted.
- In contrast to HTTP communication using `HttpConnection`, it will be possible with i-appli launch of browsers to access sites other than the site from which the i-appli was originally downloaded. However, request methods are limited to GET only, and the i-appli Runtime Environment will provide the user with the URL content when requesting permission for browser launch.

11.2 Mail Linking Launch

With Mail Linking Launch, it is possible to launch i-applis from items of mail as delivered to the user. Note that the i-applis for launching must have already been downloaded onto the mobile phone.

11.2.1 Mail Launching of i-applis

As a developer, you will be required to carry out the following two tasks to ensure that i-applis can be launched from mail.

- (1) Use the ADF's `LaunchByMail` key to declare that the i-appli is permitted to be launched from the mail as received from a specified source. It will not be possible to launch this i-appli from mail received from a source for which the corresponding permission has not been granted using the `LaunchByMail` key.
Note that the mail address as specified by the `LaunchByMail` key will be rearward-match compared with the source address for the mail, and if the content specified in the `LaunchByMail` key is all contained in the mail's source address, then launch permission will be granted for that mail.
- (2) Mail is sent from the launch-approved mail address to the mobile phone of the user having downloaded the i-appli as specified in (1) above. The mail contains information required to determine the i-appli to be launched in addition to i-appli launch parameters (i.e., application-launch data).

If mail containing i-appli-launch data is read on a mailer, a link for i-appli launching and the like will be displayed. By selecting this link, the user can launch the i-appli suitable for his or her mobile phone.

The following will provide a sample of mail containing i-appli-launch data.

```
.....
.....
.....
(End of content which may be viewed on the mailer)
--B:A
TEXT="i-appli A"
ADF="http://www.nttdocomo.co.jp/java/abc.jam"
"Param1"="i-mode"
"Param2"="i-appli"
```

Note: <CR><LF> (0x0d0a) is required at the end of each line.

The procedures for indication of i-appli-launch data are described below

- --B:A

The boundary identifier "--B:A" marks the boundary between normal mail content and the start of i-appli-launch data. The line containing only this identifier (in capital letters) and all subsequent lines will be assumed to be i-appli-launch data. It will not be possible for the user to directly view i-appli-launch data when the corresponding item of mail is being read. Furthermore, this i-appli-launch data will not be included when forwarding, or replying to, this type of mail.

- TEXT key

The `TEXT` key is used to specify a string to be displayed as the link for i-appli launching. In the above example, the character string "i-appli A" will be displayed when the mail is viewed, and by selecting this link, the user can launch the corresponding i-appli.

- ADF key

The `ADF` key is used to specify the URL of the ADF corresponding to the i-appli. When an i-appli has been downloaded onto the mobile phone, the URL for the ADF corresponding to that i-appli will be recorded. With Mail Linking, launching will be possible for those i-applis which reference the ADF with the same URL as that specified using the mail's `ADF` key.

- Parameter specification key

By enclosing both the name and value for a key in quotation marks, a developer can define parameters to be passed to the i-appli. Parameters specified in this way can then be acquired by the `IApplication.getParameter()` method (as for the `PARAM` tag for browser launching of i-applis).

Up to sixteen parameter specification keys may be included within a single mail item. Furthermore, the total of all of the name lengths and value lengths as specified for all parameter specification keys within a single item of mail is limited to a maximum of 255 bytes. Both the name and value for a key may be specified using Japanese text, and when this type of text is being used for parameters, length restrictions will be as applicable to SJIS representation.

The following diagram illustrates the relationship between an i-appli downloaded onto a mobile phone and the content of the i-appli-launch data from the corresponding mail.

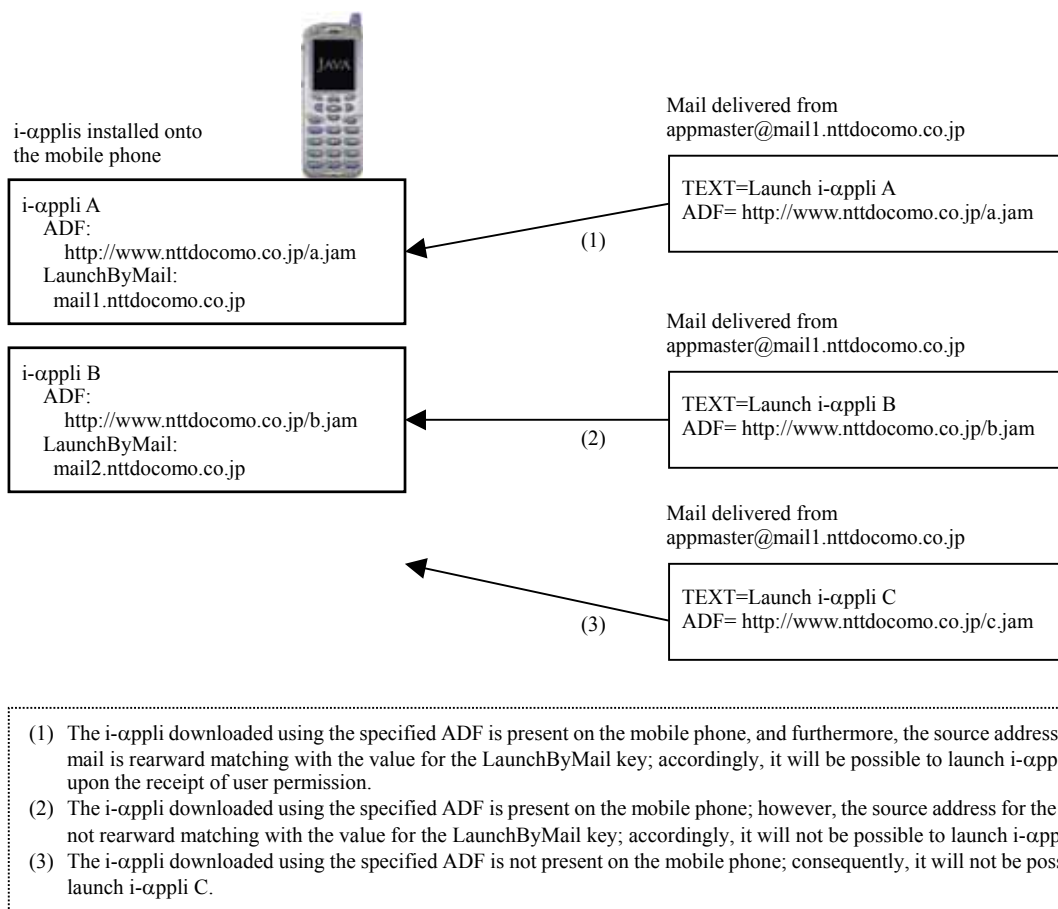


Figure 16: Downloaded i-applis and Data for i-appli Launching

Notes:

- The user of a mobile phone may use that mobile phone's settings to forbid the launching of i-applis from mail. Furthermore, before an i-appli is launched from mail, the i-appli Runtime Environment will request confirmation from the user that i-appli launching is to be permitted. If the permission of the user is not obtained through either of these methods, it will not be possible to launch the corresponding i-appli.
- When the operation of an i-appli as launched from mail comes to an end (i.e., ended by `IApplication.terminate()`), the mobile phone will return to the mailer condition which existed before launching of the i-appli (i.e., displaying the original mail). However, if the i-appli launched from mail is forced to end by the user, or the i-appli launched from mail operates or calls another native application with application linking, the state entered after i-appli ends will depend on the manufacturer.
- i-appli launch information and other attachment information (separating the mail body from other attached information, melody attachments, etc. with a boundary identifier) cannot be set in the mail at the same time. If they are set at the same time, they will be treated as invalid and ignored.
- Mail size, including i-appli launch information, must be within the restrictions of iMode mail. This feature does not support segmented incoming iMode mail, so the mail must be sent at its unsegmented size.

11.3 External Device Linking Launch

With External Device Linking Launch, an external device can instruct the mobile phone to carry out launching via the infra-red port; thus enabling an i-appli on the mobile phone to be launched. Note that the i-applis for launching must have already been installed on the mobile phone.

11.3.1 External-Device Launching of i-applis

As a developer, you will be required to carry out the following two tasks to ensure that i-applis can be launched from an external device.

- (1) Create and locate on the external device an OBEX application for instructing the mobile phone via the infra-red port to launch an i-appli. Refer to Section 10.3 for details on the conditions relevant to infra-red communication on the external device. Indication of i-appli launching from the external device is carried out by transmitting the `vTrigger` object as described below to the mobile phone.
- (2) Use the ADF's `AllowPushBy` key for the i-appli to be external-device launched to declare that said i-appli may be launched by an external device which sends a `vTrigger` object containing that application's specific launch command. In situations where the launch command declared using `AllowPushBy` and the launch command transmitted by the external device are not exact matches, the launching of that i-appli will not be possible. The `AllowPushBy` key is specified as described below. For more details regarding ADF specification methods, refer to Section 13.5.1.

```
AllowPushBy = Irda:<Startup command>
```

The mobile phone's infra-red port may be activated through menu and button operations by the user. If, when in this condition, the user aligns the external device and the mobile phone's infra-red port to be facing each other, and a `vTrigger` object is received, then an i-appli can be launched.

Specifically, `vTrigger` is an object format specified in order to provide support for external device linking launch to i-appli compatible mobile phones; furthermore, this format comprises text data which represents a combination of a number of different properties (i.e., attribute name and value pairs). By transmitting this `vTrigger` object to a mobile phone using its infra-red port, an i-appli on said mobile phone can be instructed to launch.

The following will describe the format for the vTrigger object.

```
BEGIN:VTRG
VERSION:<vTrigger format version>
ADFURL:<ADF URL for the launch i-appli>
COMMAND:<Launch command>
<Parameters>
END:VTRG
```

Note: <CR><LF> (0x0d0a) is required at the end of each line.

vTrigger is constructed in accordance with the following:

- BEGIN:VTRG

This represents the start of the vTrigger object.

- VERSION:<vTrigger format version>

This is used to specify the format version of the vTrigger object which is being used. With DoJa-2.0 Profile and DoJa-3.0 Profile, the vTrigger format version is specified as 1.0.

- ADFURL:<ADF URL for the launch i-appli>

This specifies in ASCII format (255 bytes maximum) the URL for the ADF corresponding to the application which it is desired to launch with this vTrigger.

- COMMAND:<Launch command>

This specifies in ASCII format (250 bytes maximum) the launch command. If the launch commands specified here, and the launch commands permitted by the i-appli ADF AllowPushBy key (command string specified after the "irda:" AllowPushBy key value) do not completely agree, i-appli will not be able to launch.

- <Parameters>

This specifies the (optional) parameters to be passed to the launched i-appli. Each parameter must adopt the following format.

```
PARAM [;<Encode specification>][;<Character set specification>]:
      <Parameter name>;<Parameter value>
```

*: Items in square brackets may be omitted. Furthermore, switching of the positions of the encode specification and the character set specification is permitted.

The parameter name and parameter value are used to specify the parameter which may be acquired by the `IApplication.getParameter()` upon launching of the i-appli. Up to sixteen individual <Parameter> items may be specified for a single vTrigger object; however, the combined length of all <Parameter> names and values is restricted to 255 bytes.

Encode specification and character set specification is carried out as described below.

- Encode

Specification is carried out in the format `ENCODE = <Encode name>`. The encode name can be either `QUOTED-PRINTABLE` or `BASE64`, and if no entry is made, then it will be assumed that encode specification is not being performed.

- Character set

Specification is carried out in the format `CHARSET = <Character set name>`. With DoJa-2.0 Profile and DoJa-3.0 Profile, the only valid character set name is `SHIFT_JIS`; furthermore, even if no character set is specified, it will be assumed that `SHIFT_JIS` is to be adopted.

Note that the body content for the vTrigger object must be specified using 7-bit code. If Japanese text (i.e., `SHIFT_JIS`) is to be used for parameter names or values, please ensure that either `QUOTED-PRINTABLE` or `BASE64` format is used for encoding. Any Japanese parameters which are encoded will be automatically decoded by the i-appli Runtime Environment in accordance with the specified encoding type, and furthermore, these will be passed to the `IApplication.getParameter()` method after having first being converted into a Java string.

- END:VTRG

This represents the end of the vTrigger object.

The following presents an example of vTrigger object representation.

```
BEGIN:VTRG
VERSION:1.0
ADFURL:http://www.nttdocomo.co.jp/java/a.jam
COMMAND:startup Application A
PARAM:Param1;i-mode
PARAM;ENCODING=BASE64;CHARSET=SHIFT_JIS:Param2;gomDQYN2g4o=
END:VTRG
```

The term “transmission of a vTrigger object in the OBEX layer” refers to the setting of content for the vTrigger object in the Body header, and then the transmission of this using a PUT operation. In response to instruction for i-appli launching from an external device, a mobile phone will return an OBEX response code as described below in accordance with a judgement on the possibility of launching.

Response code	OBEX standard	Standard for external-device linking
0xA0	OK, Success	Launching is possible for the specified i-appli.
0xC4	Not Found	The specified i-appli does not exist.
0xE1	Database Locked	External-device launching is forbidden for the specified i-appli; alternatively, the AllowPushBy key's launch command and the launch command as received with the vTrigger object do not match.

The i-appli will not yet have been launched on the mobile phone upon return of the response code 0xA0 to the external device. i-appli launching will take place after the mobile phone returns the response code and infra-red communication has been completed.

The following diagram illustrates the relationship between an i-appli downloaded onto a mobile phone and the content of the vTrigger object.

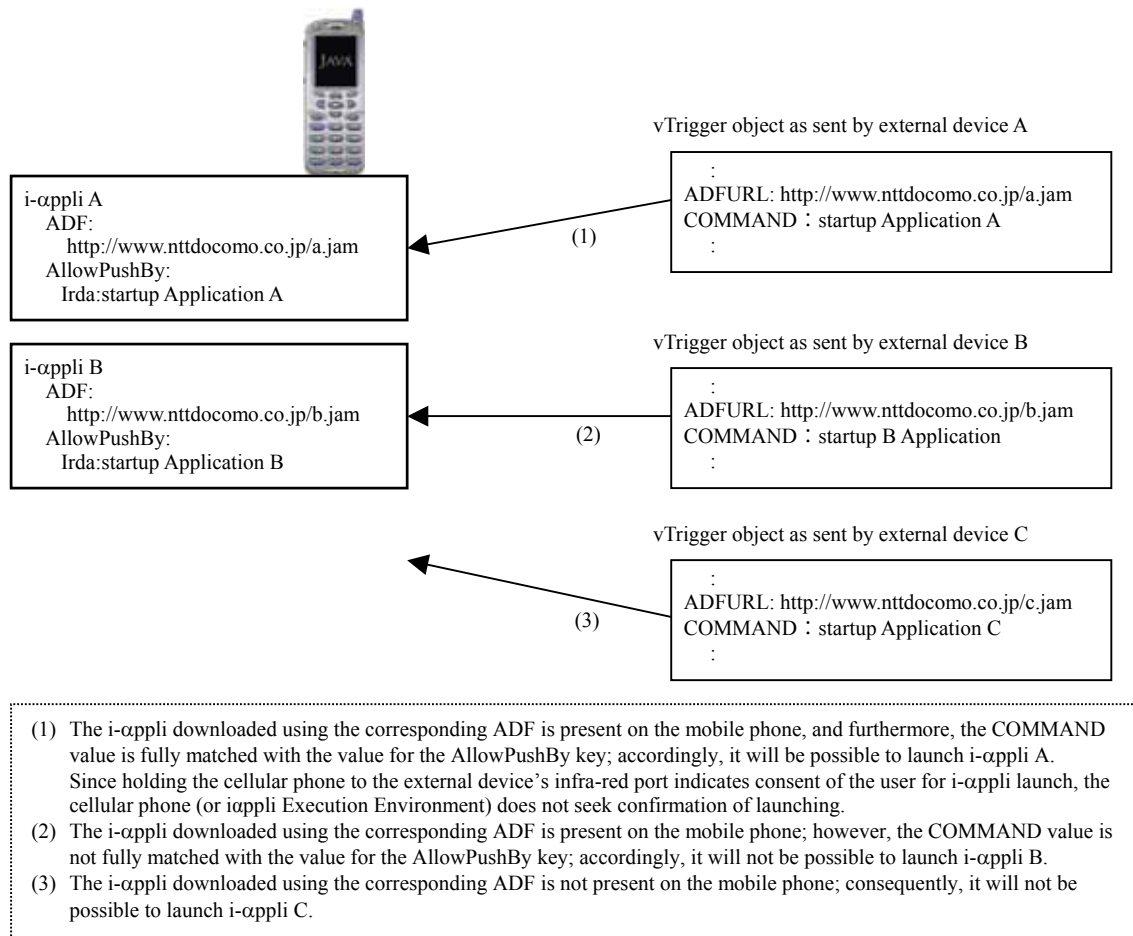


Figure 17: Download i-applis and vTrigger Object Content

Notes:

- The user of a mobile phone may use that mobile phone's settings to forbid the launching of i-applis from an external device. When this is done, no i-applis will be launched, even if a vTrigger object is received.
- In order to facilitate the receipt of launch commands (i.e., vTrigger objects) from an external device, it will be necessary for the user to activate the mobile phone's infra-red port in advance. Different manufacturers employ different methods to activate the infra-red ports.
- When the operation of an i-appli as launched from an external device comes to an end, the mobile phone will return to the stand-by condition. However, when a user implements forced shutdown for an i-appli launched from an external device, the condition adopted following the ending of this i-appli will depend on the manufacturer in question.
- When sending a vTrigger object from an external device to a mobile phone, please ensure that the identifier "vtg" is added to the value in that object's Name header. When the Name header of a received object contains this identifier, it will be recognized as a vTrigger object.

11.4 In i-appli Linked Launch

With i-appli linked launch, an API call from the application program can launch another i-appli. There are two modes of i-appli linked launch:

- Link mode

In the link mode, when the i-appli launches another i-appli, the i-appli to be launched can be directly specified with the URL of ADF. Further, the original i-appli can pass parameters to the launched i-appli. However, in the link mode the original i-appli and the launched i-appli both must be ones downloaded from the same host.

- Launcher mode

Unlike the link mode, in the launcher mode the original i-appli can launch an i-appli from a different download host. However, when launching the i-appli, the entry ID corresponding to the individual i-appli downloaded to the mobile phone is used to specify the i-appli, not the URL of ADF. For this reason, when launching an i-appli in the launcher mode, it is necessary to retrieve in advance the entry ID of the i-appli targeted for launching. Retrieval of i-appli entry IDs is performed with an API call, and requires user verification.

In the launcher mode, no parameters can be passed from the original i-appli to the launched i-appli.

The i-appli which is to be launched must already be installed on the mobile phone.

11.4.1 i-appli Linked Launch Using the Link Mode

The `IApplication.launch()` method is used to launch an i-appli in the link mode. When launching an i-appli in the link mode, this method's first argument (classification of the application to be launched) is `IApplication.LAUNCH_IAPPLI`, and the first item in the second argument (String array indicating launch arguments), specifies the ADF URL of the i-appli you wish to launch.

When passing parameters to the launched i-appli, the second item of the second argument's String array element (array index: 1 or greater) is used. Parameters passed from the original i-appli can be retrieved in the launched i-appli using the `IApplication.getParameter()` method. One parameter is composed of two array elements indicating the variable name and value.

* Parameter settings for the original i-appli

```
String[] args = new String[5];
// Set ADF URL of i-appli to be launched
args[0] = "http://www.nttdocomo.co.jp/app.jam"

// Parameter settings
args[1] = "Param1"; // 1st parameter's name
args[2] = "i-mode"; // 1st parameter's value
args[3] = "Param2"; // 2nd parameter's name
args[4] = "i-appli"; // 2nd parameter's value

// i-appli linked launch (link mode)
IApplication.getCurrentApp().launch(
    IApplication.LAUNCH_IAPPLI, args);
```

* Parameter retrieval in the launched i-appli

```
String param1, param2;
// "i-mode" stored in param1
param1 = IApplication.getCurrentApp().
    getParameter("Param1");

// "i-appli " stored in param2
param1 = IApplication.getCurrentApp().
    getParameter("Param2");
```



Figure 18: Passing and retrieval of parameters from original i-appli to launched i-appli in the link mode

A maximum of 16 parameters (16 sets of parameter names and values) can be specified. Also, the combined length of a specified parameter's names and values is limited to 255 bytes, maximum (number of bytes is evaluated in terms of default encoding).

Regarding i-appli which perform i-appli linked operations, it is necessary to declare that i-appli linked operations will be performed in the ADF's LaunchApp key (set value to "yes"). In i-appli for which this declaration is not made, i-appli linked operations cannot be performed.

Notes:

- The original i-appli's ADF URL and the ADF URL host name of the i-appli to be launched must be the same. Also the original i-appli cannot launch an i-appli that has the same ADF URL.
- This functionality cannot be used from i-appli which launch immediately after download, or inactive mode stand-by applications.
- When the `IApplication.launch()` method is called, the original i-appli enters a suspended state. After this, the i-appli runtime environment will launch the new i-appli if the user agrees to the i-appli launch verification request. If the user does not agree to the i-appli launch, the original i-appli resumes, and the `IApplication.launch()` method returns control to the caller without generating an exception. Also, when the user agrees and the new i-appli launches, the original i-appli will end.
- When the launched i-appli completes operating, the mobile phone returns to the stand-by state. However, if the launched i-appli is forcibly terminated by the user, the state entered after the i-appli terminates varies depending on the manufacturer.
- Even if the i-appli to be launched is a stand-by application, when launching with an i-appli linked operation, it will be treated as a normal, not a stand-by, operation.
- If the original i-appli is a trusted i-appli, then the i-appli to be launched must also be a trusted i-appli.

11.4.2 i-appli Linked Launch with the Launcher Mode

When performing i-appli linked launch with the launcher mode, the developer must implement the following two items of handling in the i-appli which will initiate the launching (called the “launcher i-appli” below). Also, just as in the link mode, when performing i-appli linked operations in the launcher mode, it is necessary to declare that i-appli linked operations will be performed in the ADF LaunchApp key (set value to “yes”).

(A) Retrieve entry ID of i-appli that will be launched

To launch an i-appli in the launcher mode, the entry IDs, which are assigned on a 1 to 1 basis to each i-appli downloaded to the mobile phone, are used. Entry IDs are random integer values, and obtaining valid entry IDs requires going through an i-appli selection process performed by the user (if the application program specifies an invalid entry ID and attempts to launch an i-appli, the i-appli Runtime Environment displays a warning dialog to the user). The i-appli selection process is performed using the `com.nttdocomo.system.ApplicationStore` class. When the static method `selectEntry()` of this class is called, i-appli suspends and an i-appli list is displayed on the screen. When the user selects an i-appli from that list, i-appli resumes, and the `selectEntry()` method returns an `ApplicationStore` object for the selected i-appli. If the `getId()` method is called on the returned `ApplicationStore` object, the entry ID for the selected i-appli can be retrieved.

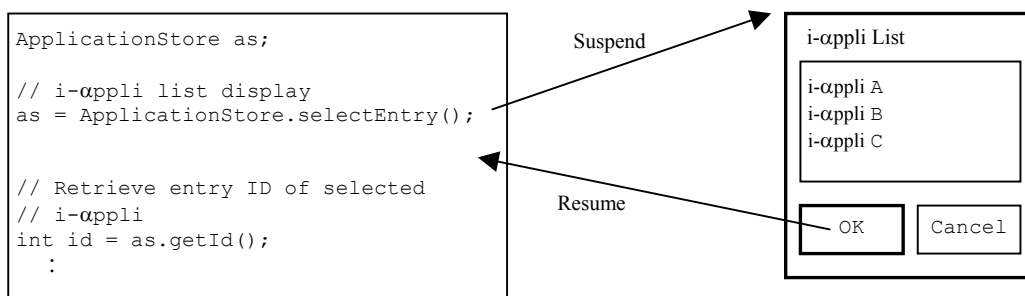


Figure 19: i-appli entry ID retrieval

The entry ID assigned to an i-appli is valid as long as that i-appli exists on the mobile phone; even if the i-appli is updated, the ID will be passed on to the updated i-appli.

Information regarding user-selected i-appli other than entry IDs (for example, i-appli names) cannot be retrieved.

(B) Launching an i-appli by specifying the entry ID

The `IApplication.launch()` method is used to specify an entry ID and launch the i-appli in the launcher mode. The first argument of the `launch()` method (classification of the application to be launched) is `IApplication.LAUNCH_AS_LAUNCHER`, while the first element of the second argument (`String` array indicating launch arguments) is the entry ID (character string).

Unlike the link mode, in the launcher mode no parameters can be passed to the i-appli being launched.

Notes:

- This functionality cannot be used from an i-appli which launches immediately after download, or inactive state stand-by applications.
- When the `IApplication.launch()` method is called, the original i-appli enters a suspended state. After this, the i-appli runtime environment will launch the new i-appli if the user agrees to the i-appli launch verification request. If the user does not agree to the i-appli launch, the original i-appli resumes, and the `IApplication.launch()` method returns control to the caller without generating an exception. Also, when the user agrees and the new i-appli launches, the original i-appli will end.
- When the launched i-appli completes operating, the mobile phone returns to the stand-by state. However, if the launched i-appli is forcibly terminated by the user, the state entered after the i-appli terminates varies depending on the manufacturer.
- Even if the i-appli to be launched is a stand-by application, when launching with an i-appli linked operation, it will be treated as a normal, not a stand-by, operation.
- There are cases in which launching with the launcher mode is not appropriate because of the i-appli's content, such as stand-by applications that do not expect to be launched normally. It is possible for an i-appli of this kind to refuse linked operation in the launcher mode. In this case, refusal of linked operation is declared in the ADF `LaunchByApp` key (value set to "deny"). An i-appli that has been set in this way will not be the object of selection in an i-appli selection process.
- In cases where, during the i-appli selection process, no selectable i-appli exist on the mobile phone, or the user cancels the selection operation, the `ApplicationStore.selectEntry()` method returns null.
- A newly downloaded i-appli is immediately given an entry ID, and that ID remains valid (even if the i-appli is updated) so long as the i-appli is not deleted. If the downloaded i-appli is deleted, then downloaded again, it will be treated as though it were newly downloaded, and assigned a new random entry ID.

11.5 i-appli Update Functionality Linked Operations

In linked operation of i-appli update functionality, the JAM i-appli update functionality can be launched from i-appli. The JAM i-appli update functionality is the feature corresponding to the “version update” item provided in the mobile phone’s i-appli list submenu. Using this functionality, the i-appli itself can encourage the user to update the i-appli.

To launch the i-appli update functionality, the `IApplication.launch()` method is used. The first argument of the `launch()` method (classification of the application to be launched) is `IApplication.LAUNCH_VERSIONUP`, the second argument (String array indicating launch arguments) is null.

When i-appli update functionality is launched, the mobile phone behaves as follows:

- (1) The `IApplication.launch()` method is called from the application program. When this method is called, the i-appli suspends.
- (2) The i-appli Runtime Environment confirms that packet communication is available (i.e. whether the “i” mark is lit or flashing on the mobile phone’s display); if packet communication is available, the user is asked to confirm the i-appli update. If packet communication is unavailable, or the user does not accept the update, i-appli resumes and the `IApplication.launch()` method returns to the caller without generating an exception.
- (3) If the user accepts in (2), the i-appli terminates, the i-appli update functionality launches, and a download is performed on the ADF corresponding to the calling i-appli. The i-appli will not be updated if as a result of downloading the ADF, it is decided that no update is needed (i.e. the ADF `LastModified` key has not been updated).
- (4) The JAR file is downloaded, and the i-appli residing on the mobile phone is updated. After this, if ADF contents requiring user confirmation (decision on HTTP communications, etc.) were updated, the user will be taken to the individual setting screens for these.
- (5) When the JAR file download and individual settings are complete, the updated i-appli will automatically restart. If a stand-by application that was launched in stand-by is updated with this functionality, it will restart in stand-by.

Notes:

- Linked operation of the i-appli update functionality cannot be performed from an i-appli which launches immediately after download, or inactive state stand-by applications.

11.6 Calling Voice-Call Functionality

In calls to voice-call functionality, it is possible to call the mobile phone's dialer functionality. It is also possible to reference the individual identification information carried by that mobile phone's hardware.

11.6.1 Dialing from i-appli

The `com.nttdocomo.util.Phone` method's `call()` class is used to call voice-call dialer functionality from i-appli. By calling this method, specifying a character string, representing the phone number, as an argument, the voice-call dialer functionality is called.

The numbers 0~9, '#', '*', ',', ' ' (pause: 1 second), and '/' (pause: wait for key input) can be specified in the telephone number character string. Pauses cannot be specified back-to-back in a telephone number character string. Further any '-' characters in the telephone number character string are ignored.

An i-appli that uses the voice-call dialer functionality must declare that it does so in the ADF `UseTelephone` key (value "call" is set). i-appli that do not perform this declaration cannot use the voice-call dialer functionality.

Notes:

- Inactive state stand-by applications cannot launch voice-call dialer functionality.
- When the `Phone.call()` method is called, i-appli enters a suspended state. After this, the i-appli Runtime Environment will operate the dialer if the user agrees to the voice-call confirmation request. When the voice call ends, i-appli resumes, and the `call()` method returns to the caller.

11.6.2 Referencing Individual Identification Information from i-appli

Individual identification information is a unique ID individually assigned to each device. There are two kinds (see below) of individual identification information, and the same value will never be assigned to multiple individuals.

- Mobile phone unit's identification information (serial number)`Phone.TERMINAL_ID`
- For mobile phones equipped with a UIM card, the UIM card identification information (serial number)`Phone.USER_ID`

This information alone cannot be utilized to identify individual users (there are no means to do so). However, by using this information along with a suitable user management system, system management, such as managing server access, can be easily achieved.

To retrieve individual identification information, the static method `getProperty()` of the `com.nttdocomo.util.Phone` class is used. In this method's argument, one of the above-mentioned pieces of individual identification information is specified for retrieval.

For i-appli to reference individual identification information, the ADF's `GetUtn` key must be used to declare that individual identification information will be referenced. The mobile phone's individual identification information and the UIM card serial number can be retrieved by specifying the values "terminalid" or "userid" respectively, in the `GetUtn` key. When referencing both, separate the two with a comma.

Notes:

- Referencing of individual identification information cannot be performed from an i-appli which launches immediately after download.
- UIM card individual identification information cannot be retrieved in mobile phones not equipped with a UIM card.

11.7 Calling Phonebook Management Functionality

By calling phonebook management functionality, the mobile phones native phonebook management functionality can be called from i-αpli, and after receiving user permission, new phonebook entries (set of phone number, name, mail address, etc.) or new phonebook groups can be added.

Regarding an i-αpli which uses this functionality, it is necessary to declare this functionality will be used in the ADF `AccessUserInfo` key (set value to “yes”).

11.7.1 Adding New Phonebook Groups

In order to add new phonebook groups, the static method `addEntry()` of the `com.nttdocomo.system.PhoneBookGroup` class is used. In this method's arguments, the name of the phonebook group to be added, or null is specified.

When adding a new phonebook group, the mobile phone behaves as follows:

- (1) The `PhoneBookGroup.addEntry()` method is called from the application program. When this method is called, i-αpli suspends.
- (2) The native user interface screen for adding new phonebook groups is launched. In this screen, the user can confirm or cancel the addition of the new phonebook group from i-αpli, and can also change the group name.
- (3) When the user confirms the new phonebook group entry, the new phonebook group is recorded using the name that was set, i-αpli resumes, and the `addEntry()` method returns to the application program. At this time, this method returns one entry ID for each new group name recorded on the mobile phone. The entry ID can be used to specify phonebook entry groups in the recording of a new phonebook entry, as described below.

If the user cancels the new phonebook group addition, no exception occurs and this method returns -1.

Notes:

- Addition of new phonebook groups cannot be performed from an i-αpli which launches immediately after download, or inactive state stand-by applications.
- Depending on the manufacturer, registering new phonebook groups may not add groups, but rather change the names of preset groups in the mobile phone.

11.7.2 Recording New Phonebook Entries

To record new phonebook entries, the static method `addEntry()` of the `com.nttdocomo.system.PhoneBook` class is used. This method supports the following information in recording phonebook entries.

- Name
- Kana (phonetic) spelling of name
- Phone number (multiple numbers can be specified)
- E-mail address (multiple addresses can be specified)
- Target phonebook group (group name or group entry ID)

There are two ways, listed below, to give this information, in the form of parameters, to the `addEntry()` method.

- (1) Enumerate these items as arguments of the `addEntry()` method.
- (2) Create a `com.nttdocomo.system.PhoneBookParam` object encapsulating each item to be designated in the phonebook entry, and specify it as an argument of the `addEntry()` method.

When adding a new phonebook entry, the mobile phone behaves as follows:

- (1) The `PhoneBook.addEntry()` method is called from the application program. When this method is called, i-appli suspends.
- (2) The native user interface screen for adding new phonebook entries is launched. In this screen, the user can confirm or cancel the addition of the new phonebook entries from i-appli, and can also change each item of the entry.
- (3) When the user confirms the new phonebook entry, the new phonebook entry is recorded using the name that was set, i-appli resumes, and the `addEntry()` method returns to the application program.

If the user cancels the addition of the new phonebook entry, no exception occurs and the method returns to the application program.

Notes:

- Addition of new phonebook entries cannot be performed from an i-applis which launches immediately after download, or in-active state stand-by applications.
- The number of phone numbers and mail address that can be recorded in one entry varies depending on the model of phone, but the minimum for all models is 3 of each.
- Depending on the model of phone, the phonebook entry's Name and Kana items can be divided into last and first names. In such models, when not using the `PhoneBookParam` object, the information will be set to the last name for both Name and Kana.

If using the `PhoneBookParam` object, the information can be divided into last and first names for Name and Kana.
- If the phonebook group name is set to null, or the phonebook group entry ID is set to -1, this means the phonebook group is not specified. In this case, the phonebook entry will be recorded in the usual place for that mobile phone.
- When a phonebook group entry ID is specified and a phonebook entry added, an exception will occur if the phonebook group corresponding to the specified entry ID does not exist. In contrast to this, when a phonebook group name is specified and a phonebook entry is added, if no phonebook group of that name exists, the i-appli Runtime Environment will ask the user whether or not to create that phonebook group.

11.8 Calling Bookmark Management Functionality

By calling bookmark management functionality, the mobile phone's native bookmark management functionality can be called from i-appli, and after receiving user permission, new bookmarks (URL and title) can be added.

Regarding i-appli which use bookmark management functionality, it is necessary to declare this functionality will be used in the ADF `AccessUserInfo` key (set value to "yes").

In order to add new bookmarks, the static method `addEntry()` of the `com.nttdocomo.system.Bookmark` class is used. In this method's arguments, the title of the bookmark to be added, and URL are specified.

When adding a new bookmark, the mobile phone behaves as follows:

- (1) The `Bookmark.addEntry()` method is called from the application program. When this method is called, i-appli suspends.
- (2) The native user interface screen for adding new bookmarks is launched. In this screen, the user can confirm or cancel the addition of the new bookmark from i-appli, and can also change the URL and title that will be recorded.
- (3) When the user confirms the new bookmark entry, the new bookmark is recorded with the details that were set, i-appli resumes, and the `addEntry()` method returns to the application program.

If the user cancels the addition of the new bookmark, no exception occurs and the method returns to the application program.

Notes:

- Addition of new bookmarks cannot be performed from an i-applis which launches immediately after download, or inactive state stand-by applications.
- The scheme of URL's that can be added as bookmarks must be HTTP or HTTPS.

11.9 Calling Schedule Management Functionality

By calling schedule management functionality, the mobile phone's native schedule management functionality can be called from i-appli, and after receiving user permission, new schedules (description and time/date) can be added.

Regarding in i-appli which uses schedule management functionality, it is necessary to declare this functionality will be used in the ADF `AccessUserInfo` key (set value to "yes").

In order to add new schedules, the static method `addEntry()` of the `com.nttdocomo.system.Schedule` class is used. In this method's arguments, the description of the schedule to be added, time/date and alarm sound on/off are specified.

The `ScheduleDate` class of the `com.nttdocomo.util` package is used to specify the schedule's time/date. The `ScheduleDate` class can represent one-time only schedules, or a daily, weekly, monthly and yearly periodic schedules (setting of the schedule's time/date specified type).

When adding a new schedule, the mobile phone behaves as follows:

- (1) The `Schedule.addEntry()` method is called from the application program. When this method is called, i-appli suspends.
- (2) The native user interface screen for adding new schedules is launched. In this screen, the user can confirm or cancel the addition of the new schedule from i-appli, and can also change the description, etc. that will be recorded.
- (3) When the user confirms the new schedule entry, the new schedule is recorded with the details that were set, i-appli resumes, and the `addEntry()` method returns to the application program.

If the user cancels the addition of the new schedule, no exception occurs and the method returns to the application program.

Notes:

- Addition of new schedules cannot be performed from an i-appli which launches immediately after download, or inactive state stand-by applications.
- In registering schedules, the date/time specification types available vary depending on the manufacturer. The common date/time specification type for all models of phone is “one-time only”. The `Schedule.getSupportedTypes()` method can be used to find out what date/time specification types a given model can use.

11.10 Calling Image Data Management Functionality

Mobile phones that support DoJa-3.0 Profile have an image management area called MyPicture (the menu item name “Multimedia” on FOMA mobile phones). By calling image data management functionality, the mobile phone’s native image data management features can be called from i-appli, and images can be input to/output from the MyPicture area. The `com.nttdocomo.system.ImageStore` class is used to call image data management functionality.

Regarding i-appli which use this functionality, it is necessary to declare this functionality will be used in the ADF `AccessUserInfo` key (set value to “yes”). Also, i-appli cannot access images downloaded and saved using the browser. Images i-appli can access are limited to the following:

- Images shot and saved with the mobile phone’s camera
- Images saved by i-appli (including another i-appli)
- Images obtained by the mobile phone from an external device with the native infrared communication tool.
- The mobile phone’s factory preinstalled images
- In mobile phones equipped with external memory, such as a memory card, images moved from external memory to the MyPicture area.

11.10.1 Adding New Image Data

In order to add new image data, the static method `addEntry()` of the `ImageStore` class is used. In this method’s arguments, the `MediaImage` object indicating the image to be added is specified.

When adding new image data, the mobile phone behaves as follows:

- (1) The `ImageStore.addEntry()` method is called from the application program. When this method is called, i-appli suspends.
- (2) The native user interface screen for adding new image data is launched. In this screen, the user can confirm or cancel the addition of the new image data from i-appli. The user can also confirm what kind of image data i-appli is trying to add.
- (3) When the user confirms the new image data entry, the new image data is recorded, i-appli resumes, and the `addEntry()` method returns to the application program. At this time, this method returns one entry ID for each new piece of image data recorded in the mobile phone. The entry ID can be used in specified image data ID loading in order to load image data without user operations, as described below.

If the user cancels the addition of the new image data, no exception occurs and the method returns -1.

Notes:

- Addition of new image data cannot be performed from an i-appli which launches immediately after download, or inactive state stand-by applications.
- The `MediaImage` object specified in the `addEntry()` method must be in a usable state (`use()` method must have been called).
- Image data recorded in the MyPicture area from i-appli can be stored outside the mobile phone with native mobile phone features (infrared communications, sending iShot mail, etc.). If it is not desirable to have the image data stored outside the mobile phone, it should be saved in ScratchPad not the MyPicture area.

11.10.2 Image Data Selection Loading

Image data selection loading is functionality that provides the image data selected by the user from a list of images displayed on the screen, every time there is a request (method call) for image data from the application program. In order to perform image data selection loading, the static method `selectEntry()` of the `ImageStore` class is used.

When performing image data selection loading, the mobile phone behaves as follows:

- (1) The `ImageStore.selectEntry()` method is called from the application program. When this method is called, i-appli suspends.
- (2) The native user interface screen for selecting image data is launched. In this screen, the user selects the image data to which i-appli will be given access. The user can also cancel, terminating the operation
- (3) When the user commits an image data entry selection, i-appli resumes, and the `selectEntry()` method returns to the application program. At this time, this method returns an `ImageStore` object for the selected image data.

If the user cancels the image data selection, no exception occurs and the method returns null.

A byte image of the selected image or `MediaImage` object can be retrieved from the `ImageStore` object returned by the `selectEntry()` method.

Also, an entry ID for the image data can be retrieved from this `ImageStore` object, which can be used in specified image data ID loading in order to load this image data again without user operation, as described below.

Note:

- Image data selection loading cannot be performed from an i-appli which launches immediately after download, or inactive state stand-by applications.

11.10.3 Specified Image Data ID Loading

The image data entry ID is a unique integer assigned by the system at the time the image is stored in the MyPicture area; it does not change until the image data is erased or overwritten by another image. If i-appli retrieves the image data's valid entry ID while the image data is being newly added or during selection loading, i-appli can then access the image data without going through a user operation. In order to perform specified image data ID loading, the static method `getEntry()` of the `ImageStore` class is used.

If the `getEntry()` method is called, with the entry ID of the image data specified as an argument, the `ImageStore` object corresponding to that image is returned. Just as when performing image data selection loading, the image's byte image or `MediaImage` object can be retrieved from this `ImageStore` object.

Note:

- Specified image ID Loading cannot be performed from an i-appli which launches immediately after download. However, because this functionality requires no user operation, it can be used by inactive state stand-by applications.

11.11 Calling Camera Functionality

Mobile phones that support DoJa-3.0 Profile are normally equipped with camera functionality. Calls to the camera functionality provide a means for accessing this functionality from i-appli. In models that support moving images, handling of moving as well still images may be supported.

The `com.nttdocomo.device.Camera` class is used to call camera functionality.

11.11.1 Camera Control

Camera control functionality is provided by the `com.nttdocomo.device.Camera` class. In some cases, the mobile phone may be equipped with multiple camera devices, but each device is identified with its camera ID (integers starting from 0, permanently assigned to individual devices). The application program can retrieve a camera object by specifying a camera ID in the `Camera.getCamera()` method.

By using this functionality, i-appli can control the following with respect to the camera object.

- Execution of camera device settings
- Operation of native camera features
- Retrieving photographed images

(A)Execution of camera device settings

There are three camera device settings: device attribute, photograph size, and frame image.

Device attribute settings are performed with the `Camera.setAttribute()` method. Attributes such as the following are provided.

- Sequential shot mode (only enabled when shooting still images)
- Picture quality
- Microphone volume (only enabled when shooting moving images)

What attributes are supported by the camera device vary depending on the model of phone, but the application program can determine the attributes supported by the mobile phone with the `Camera.isAvailable()` method. Further, the current value set for each attribute can be learned with `Camera.getAttribute()`.

The `Camera.setImageSize()` method is used to set the photograph size. The photograph size height and width are given in the number of dots. If the specified photograph size is not supported by the mobile phone, the system will correct the specified size to a supported size. Additionally, the frame image is set using the `Camera.setFrameImage()` method.

Although the camera device may support multiple photograph sizes, it may not be possible to take pictures with frames for all sizes. The below methods can be used to retrieve information regarding the camera device's photograph sizes.

- `Camera.getAvailablePictureSizes()`: retrieves sizes available for shooting still images
- `Camera.getAvailableMovieSizes()`: retrieves sizes available for shooting moving images
- `Camera.getAvailableFrameSizes()`: retrieves sizes available for shooting still images with frames

(B) Operation of native camera features

The application program can operate the camera functionality in still image mode by calling the `Camera.takePicture()` method. In models that support the taking of moving images from i-appli, the camera functionality can be operated in moving image mode by calling the `Camera.takeMovie()` method.

When camera functionality is operated, i-appli execution is interrupted and the camera operation screen is shown on the display. The actual photographing depends completely on user operations, and the application program cannot operate the shutter, etc..

When the user exits the camera operation screen, i-appli execution resumes. If the user saves the photographed image when exiting the camera function, that image will be held in the camera object.

Note:

- Operation of camera features cannot be performed from inactive state stand-by applications.

(C) Retrieving photographed images

Images photographed with the camera feature are held in the camera object until the next operation of the camera feature, the i-appli ends, or it is explicitly destroyed with the `Camera.disposeImages()` method, and can be referenced from the application program in the form of a media image or byte stream.

To retrieve an image held in the camera object in a media image format, the `Camera.getImage()` method is used.

To retrieve the image in a byte stream format (JPEG for still images, iMotion format for moving images), the `Camera.getInputStream()` method is used. The index for the image stored in the camera object is specified in the arguments for this method. The image's index is normally specified as 0, but when still image sequential shots are being taken, the desired image can be specified by its index.

11.11.2 Exchanging Photographed Images Between Mobile Phones

Among mobile phones that support DoJa-3.0 Profile, there are models which support the exchange of photographed images between mobile phones via the infrared port as a feature of the mobile phone itself. Because this feature is implemented based on OBEX External-Connection functionality, image exchanges can be conducted between i-appli which use OBEX External-Connection functionality and this native application.

When native applications exchange images, they use an IrMC vNote version 1.1 compliant object. However, because this feature sends additional information needed for the image exchange, the vNote object properties are extended.

The details of the object properties used in this feature are indicated below.

Item No.	Property	Value	Remarks
1	VERSION	1.1	vNote version
2	X-DOCOMO-TYPE	JPEG or GIF	Image data format classification
3	X-DOCOMO-SIZE	<width>x<height>	Image size (Width x Height) The "X" is an alphabetic half-width capital letter. (ASCII code 0x58)
4	X-DOCOMO-FILESIZE	<filesize>byte	Image file size (in bytes)
5	X-DOCOMO-FILENAME	See remarks	Filename (half-width alphanumeric character string, within 24 bytes, including extension)
6	SUMMARY	See remarks	Image title (Character string using QUOTED-PRINTABLE encoding, includes SHIFT-JIS, within 18 bytes)
7	DATE	See remarks	Date/time of when taken or edited (UTC notation)
8	X-DOCOMO-BODY	See remarks	Encoded in Base64 format, JPEG or GIF data. Add <CR><LF> to the end of this property value.

Below is an example of a vNote object for exchanging image data.

```
BEGIN:VNOTE
VERSION:1.1
X-DOCOMO-TYPE:JPEG
X-DOCOMO-SIZE:120X120
X-DOCOMO-FILESIZE:3000byte
X-DOCOMO-FILENAME:picture_001.jpg
SUMMARY;CHARSET=SHIFT_JIS;ENCODING=QUOTED-PRINTABLE:=89=E6=91=9C
DATE:20020715T153530Z
X-DOCOMO-BODY;ENCODING=BASE64:<Image data encoded in Base64 format+<CR><LF>>
END:VNOTE
```

* Include a <CR><LF> (0x0d0a) at the end of each line. In the X-DOCOMO-BODY, there will be two consecutive <CR><LF>, including the <CR><LF> at the end of the data.

When sending image data from i-αpli to the native application, the native application operates as an OBEX server. The i-αpli acts as an OBEX client, and the image data (vNote object) is sent using a PUT operation.

When sending image data from the native application to i-αpli, the native application operates as an OBEX client. The i-αpli acts as an OBEX server, and the image data, (vNote object) sent using a PUT operation from the native application, is received.

Notes:

- In exchanging images, multiple transmissions/receptions of vNote objects cannot be performed. Only one piece of image data can be sent per operation.
- In DoJa-3.0 Profile, the size of the data that i-αpli can send/receive using OBEX External-Connection functionality is limited to 100KB (30KB in DoJa-2.0). When exchanging images, the entire size of the above mentioned vNote object needs to be under this limit. This is the same whether the i-αpli is sending or receiving the image.
- The X-DOCOMO-FILESIZE property specifies the file size before Base64 format encoding
- When sending image data from i-αpli to the native application, set the Name header for the object sent by the PUT operation as "mypic.vnt".
- When sending image data from i-αpli to the native application, the sizes of image which can be sent (the image height/width size indicated by X-DOCOMO-SIZE) are limited to the sizes supported by the camera of the receiving mobile phone.

Chapter 12

The Infrared Remote Control

In DoJa-3.0 Profile, APIs have been added to transmit control signals from the mobile-phone's infrared port to devices which support standard infrared remote controls. By assembling and sending the right control-signals from i-appli, these infrared remote control supported devices can be controlled externally.

Notes:

- A primitive API for assembling and sending the infrared signal's physical wave pattern is provided in DoJa-3.0 Profile's infrared remote control functionality. However, the specific waveforms that the general infrared remote control supported device can accept as control-signals are individually designed by the supported device's manufacturer. The scope of the i-appli specification does not include virtualization/standardization of infrared remote control control-signals. It is necessary for the developer to confirm the specifications of the infrared remote control supported device to be targeted, and design the i-appli to send suitable control-signals.
- In the infrared remote control functionality of DoJa-3.0 Profile compliant mobile phones, it is assumed that devices separated from the mobile phone by about 2 to 3 meters can be controlled. However, the distance over which devices can actually be controlled will be influenced by the mobile phone's signal output power and directivity, as well as the surrounding environment and the infrared remote control supported device's photoreceptive sensitivity.

12.1 Control signal Structure

The below indicated infrared remote control control signals can be handled with DoJa-3.0 Profile. Control signals for many general home electronics sold in Japan can be expressed with the below outline. However, as mentioned above, control signal design is conducted individually by the device manufacturer, so there is no guarantee that the below applies to all devices supporting infrared remote control.

Carrier Frequency and Modulation Type:

In DoJa-3.0 Profile, the infrared remote control signal is handled as a pulse phase modulated signal (PPM signal, below the logical waveform conveyed by the carrier wave is referred to as the control-signal) carried by a carrier wave (square wave expressed by infrared transmission ON and OFF) with a 25kHz~50kHz carrier frequency. The combination of the control-signal's high intervals and low intervals express the bit values (logic 0 and logic 1).

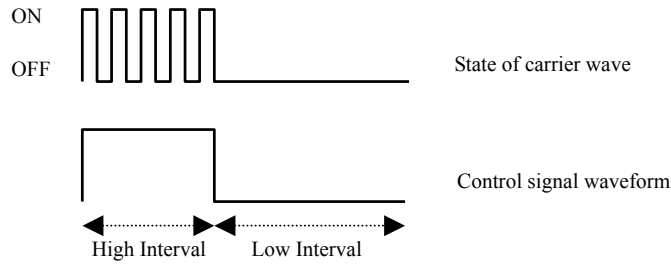


Figure 20: Composition of control signal from the carrier wave

There are two patterns of expression regarding the control signal waveform's bit value, and which method is used varies depending on the design of the infrared remote control supported device. Use of either of the below methods can be specified in i-αplli's infrared remote control functionality (length of double lines in the waveform chart are used to distinguish between logic 0 and logic 1).

	Sample Logic 0 waveform	Sample Logic 1 waveform
High first pattern Logic 0, Logic 1 expressed by Low interval's length.		
Low first pattern Logic 0, Logic 1 expressed by High interval's length.		

In the above, the following items are dependent on the design of the infrared remote control supported device; the application program can set these to values appropriate to the device.

- The High interval length and Low interval length for one carrier wave waveform (this determines the carrier frequency of the carrier wave)
- Means of expressing the bit value (High first pattern or Low first pattern)
- High interval length and Low interval length of Logic 0 on the control-signal
- High interval length and Low interval length of Logic 1 on the control-signal

Structure of Data Frame:

The infrared remote control's outgoing data is attached to the start section (control signal composed of one each of High interval and Low interval of fixed length) and stop section (control signal composed of one High interval of fixed length) of the bit data of the control signal modulated by the method described above.

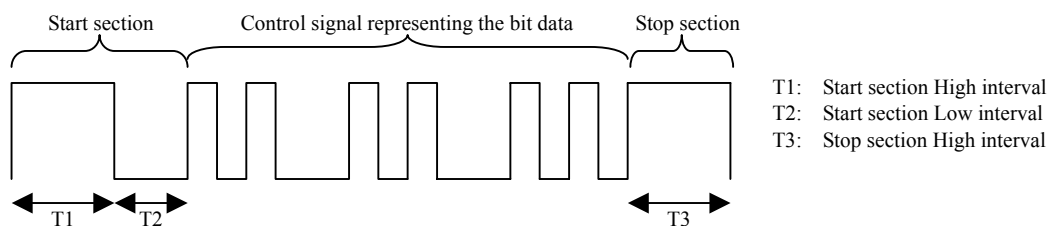


Figure 21: Data frame structure

Bit data in the figure's middle is an example of the High first pattern. Assuming $\square$ is Logic 0 and $\square$ is logic 1, the above data frame is a 6-bit bit array containing 10100

The length of the start section's High and low intervals, and the stop section's High interval are dependent on the design of the infrared remote control supported device; the application program can set these to values appropriate to the device. Also, the common maximum number of bits in the bit data that can be included in a data frame is 48 bits for all models. Data frames are expressed by the `com.nttdocomo.device.IrRemoteControlFrame` class, described in 12.2.2.

Overall Structure of Infrared Remote Control Outgoing Data:

The infrared remote control's outgoing data is expressed as a combination of data frames. In some cases completely different data frames are sent consecutively, in some cases a given data frame is sent repeatedly.

Further, in most infrared remote controls, when the provided data frames are finished sending through, they will be sent again from the first data frame. This supports, for example, cases when the volume button of a TV remote is continually pressed and the control-signal is continually sent during that period.

Repeated transmission of a data frame occurs for the time period specified when the data is sent (timeout period), or until the application program explicitly stops data transmission.

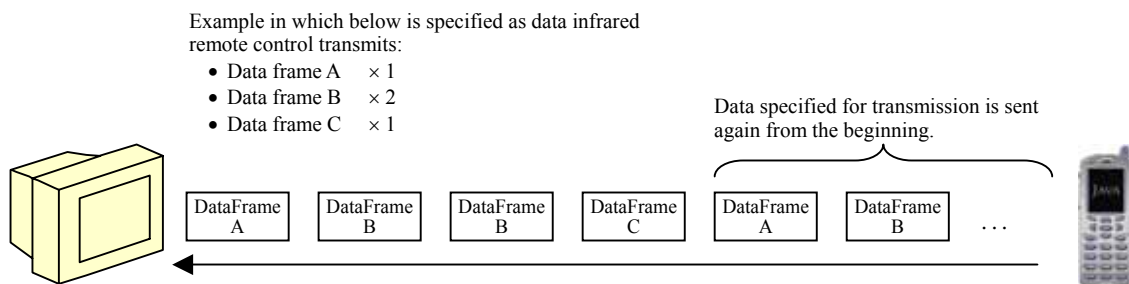


Figure 22: Example of data frame transmission from a mobile phone

The application program can set transmission ordering for the created data frame information, transmission intervals, and timeout period until the data transmission automatically stops, appropriate to the targeted device. Infrared remote control transmission data is sent using the `com.nttdocomo.device.IrRemoteControl` class, explained in 12.2.1.

12.2 The Infrared Remote Control API

In the above section, it was mentioned that the below classes were added in DoJa-3.0 Profile so that infrared remote control functionality could be used in application programs.

- `com.nttdocomo.device.IrRemoteControl`
- `com.nttdocomo.device.IrRemoteControlFrame`

The `IrRemoteControl` class acts as an emitter for transmitting control signals from the infrared port. The `IrRemoteControlFrame` class expresses the data frame that structures the outgoing data. The application program creates the data frame set in the form of an `IrRemoteControlFrame` object and sends that set from the infrared port with an `IrRemoteControl` object.

12.2.1 IrRemoteControl

The `IrRemoteControl` class has functionality for sending the data frame specified in `IrRemoteControlFrame` form. It also has methods for performing configurations spanning the entire remote control functionality.

The application uses the `IrRemoteControl` class in the following way.

- (1) An `IrRemoteControl` object is retrieved with the `IrRemoteControl.getIrRemoteControl()` method. The `IrRemoteControl` object is assigned to the mobile phone's infrared remote control port on a one-to-one basis, so an application program cannot have multiple instances of this object.
- (2) The infrared remote control's physical configuration is performed. The items set for the infrared remote control are as follows:
 - Carrier wave High interval length, Low interval length (carrier frequency) setting (`setCarrier()` method)
 - Expression pattern for Logic 0 waveform on the control signal and High interval length, Low interval length (`setCode0()` method)
 - Expression pattern for Logic 1 waveform on the control signal and High interval length, Low interval length (`setCode1()` method)

The Expression patterns for the Logic 0 waveform and Logic 1 waveform are specified by the `IrRemoteControl` class constants `PATTERN_HL` (High first) and `PATTERN_LH` (Low first).

- (3) An `IrRemoteControlFrame` object set expressing the data frame to be sent is created and transmitted using the `send()` method. In the `send()` method, the period of time for which the data frame's set will be repeatedly automatically transmitted (timeout period) can be specified. The default timeout period is 10 seconds.

The `send()` method returns to the application program immediately after receiving a transmission request; after this, data frame transmission is performed asynchronously from the application program. Data frame transmission continues repeatedly until one of the below:

- Time out period runs out
- Transmission is explicitly stopped from the application program (`stop()` method)
- Interruption due to an incoming voice-call, etc.
- Execution of HTTP communication from the application program
- Application ends (including abnormal end)

Notes:

- Application programs cannot call the `send()` method in the following cases
 - Stand-by application is in the inactive state
 - The mobile phone is set to self mode
 - i-appli is using the infrared port (infrared remote control, OBEX external-connection)
 - Packet transmission is taking place
- In setting the infrared remote control, the units of the setting values and possible value ranges for the setting depends on the method. For information on the units, etc. of each method's setting values, please refer to the API reference.
- Depending on the characteristics of the infrared device employed by the manufacturer, the waveform parameter value (High interval length, Low interval length) specified by the API may be revised to a value (unit) that can be handled by that device.

12.2.2 IrRemoteControlFrame

The `IrRemoteControlFrame` class expresses the individual data frames. The application program structures a data frame by using the constructor of this class to create an object and set the start section, bit data section and stop section.

Data frame settings are as follows.

- Duration of data frame transmission (`setFrameDuration()` method).
Specifies the duration between the beginning of this data frame's transmission, and the beginning of the next data frame's transmission.
- Number of times the data frame will be repeatedly sent (`setRepeatCount()` method)
In one data frame set, specifies the number of repetitions if the same data frame is to be sent repeatedly. If the same data frame will not be sent repeatedly, 1 is specified.
- Start section format (`setStartHighDuration()` or `setStartLowDuration()` method).
Specifies lengths for the start section's High interval and Low interval in that data frame.
- Stop section format (`setStopHighDuration()` method).
Specifies length for the stop section's High interval in that data frame.
- Bit data value (`setFrameData()` method)
Specifies values that will be sent as bit data, either in `byte` array or `long` format. The length that any model of phone can send is 48 bits per one frame.

Notes:

- In setting the data frame, the units of the setting values and possible value ranges for the setting depends on the method. For information on the units, etc. of each method's setting values, please refer to the API reference.
- Depending on the characteristics of the infrared device employed by the manufacturer, the parameter values for the start section, stop section and data frame transmission duration specified by the API may be revised to a value (unit) that can be handled by that device.

Chapter 13

Building, Testing, and Packaging the i-appli

This chapter provides an overview of the development cycle for i-appli. NTT DoCoMo offers a development environment for developing i-appli on the PC (hereinafter referred to as the i-appli Development Environment), but this chapter does not go into any detail on this development environment or supplied tools. For details on this development environment, refer to the User's Guide for the i-appli Development Environment.

The content developers are assumed to have installed the Java 2 SDK, Standard Edition (J2SE SDK version 1.3.1 or later), and be familiar with the tools and documentation that come with the J2SE SDK.

In order to install a developed i-appli on an actual mobile phone for testing, it will be necessary to set up a Web site to distribute the i-appli (a connection to the Internet will be needed). For details on distributing and i-appli, see Chapter 14.

13.1 Installing the i-appli Development Environment

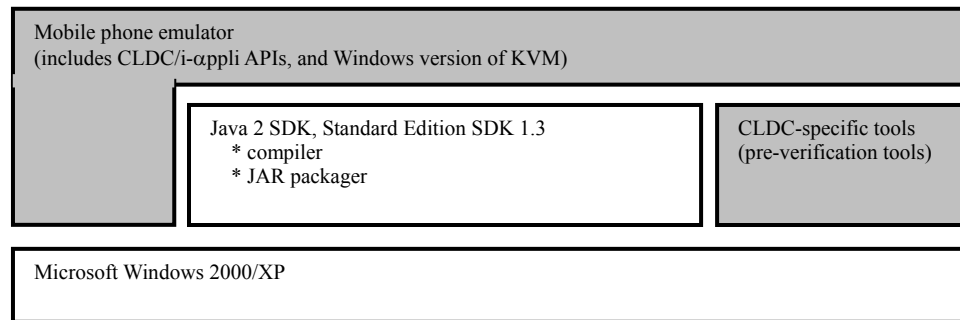
In order to use the i-appli Development Environment, it will be necessary to obtain the archive files for this environment as supplied by NTT DoCoMo, and for these to be installed on your PC. For more details regarding the installation procedure for these files, refer to the User's Guide for the i-appli Development Environment.

Note that it will also be necessary to make J2SE SDK available in order that the i-appli Development Environment may be used. Please ensure that J2SE SDK is installed in advance of the i-appli Development Environment.

13.2 Components of the i-appli Development Environment

The i-appli Development Environment is a simple development environment for developing i-appli on the PC. This environment allows you to build and emulate i-appli on your PC.

The Application Development Environment is made up of the following components:



Note: The i-appli Development Environment includes patched components.

Figure 23: Composition of the i-appli Development Environment

Below is an overview of the components included in the i-appli Development Environment:

- **Mobile phone emulator**
This is the main component of the i-appli Development Environment. Developers can use the mobile phone Emulator to build (compile, pre-verify, and package the JAR) and test i-applis. i-applis are build by calling the various build commands from the command line.
This document sometimes refers to the mobile phone Emulator as the Emulator.
- **CLDC-specific tools**
The CLDC-specific tools including pre-verification tools. Pre-verification is the act of doing some of the byte code verification work on the standard Java virtual machine beforehand (at application build time). CLDC uses the pre-verification method to reduce the cost of verification execution at application runtime. Classes which have not been pre-verified cannot be used in the i-appli Runtime Environment.
You can run the pre-verification tools independently from the command line, as well as from the i-appli build feature of the mobile phone Emulator.

[DoJa-2.0] [DoJa-3.0]

There are slight differences in the structure of the components between the development environment for DoJa-1.0 and the i-appli Development Environments offered for DoJa-2.0 and DoJa-3.0 Profiles.

13.3 Overview of the Development Cycle

The figure below illustrates the typical development cycle.

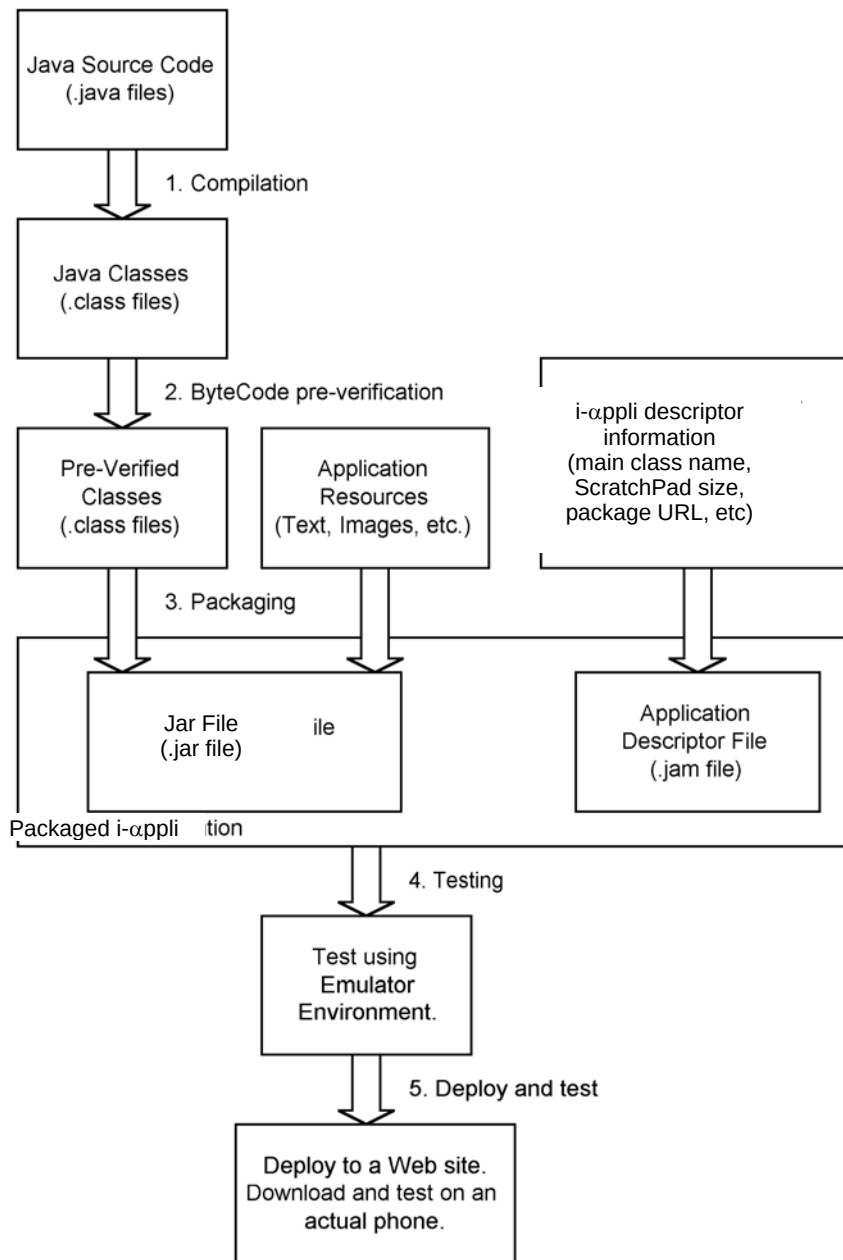


Figure 24: Development Cycle

Compilation — The Java compiler (javac) provided by the J2SE SDK is used to compile the source files into class files. The compiler also verifies that the application program is not using any packages, classes, or method that are not part of the i- α pli Runtime Environment.

Bytecode pre-verification — The Bytecode pre-verification tool provided by the Emulator performs the pre-verification process on the Java classes. When the pre-verification tool is executed, it adds additional attribute information to the class file so that the CLDC virtual machine can efficiently check Bytecode at runtime. The pre-verification tool also checks against the use of floating point operations and finalize methods in the application program.

Packaging — The J2SE SDK archive tool (jar) packages all the pre-verified class files and the resource files into a JAR file. The jar command usage is the same as the standard usage for the J2SE SDK jar command. In order to minimize the JAR file size to be stored on the mobile phone, do not specify the non-compression option.

Testing on the Emulator — Execute the application in the Emulator environment to verify that the i- α pli functions correctly.

Testing on the actual Java-compatible i-Mode — Put the i- α pli (JAR file and ADF) and download navigation HTML content on a Web site, and download it to an actual mobile phone. Run the i- α pli to verify correct functionality. Check the i- α pli's execution on the mobile phone of every target manufacturer, as there might be hardware resource capacity differences, such as memory space availability, depending on the mobile phone.

You can conduct the compilation, pre-verification, and packaging processes (the build process) independently with each tool, or you can do them all at once with the Emulator's i- α pli build feature. For details on the Emulator i- α pli build feature, see the User's Guide for i- α pli Developing Environment.

13.4 Building the Client-Server Application

If you are building a client-server i- α pli, you will need to create a server-side application to serve as the communication partner for i- α pli. The server side application can use any technology, as long as it communicates with the client-side application (i- α pli) using the HTTP protocol.

Furthermore, when i- α plis make use of OBEX external connection, it will be necessary to prepare an OBEX application for the communication partner. In cases where OBEX external connection is used on mobile phones compatible with i- α pli, both the OBEX client and the OBEX server can be developed using i- α pli API. If connecting a mobile phone with another device and if information regarding the development of communication-partner applications is required, refer to the documentation for the OBEX application development environment as used on the communication-partner platform.

13.5 Packaging the i- α pli

As shown in Figure 24, an i- α pli consists of two files: an ADF and a JAR file, which includes the JAR class files and resource files.

13.5.1 Creating the ADF

An ADF is a text file in SJIS encoding. It consists of “=” separated key/value pairs. You can edit the file using the text editor of your choice. This file specifies the following key entries (rows).

AppName: Mandatory key

i- α pli name (max. 16 bytes). When a list of i- α plis on the mobile phone is displayed, the name specified here is shown. The name can be in SJIS-encoded Japanese.

AppVer: Optional key

i- α pli version (max. 10 bytes). The developer can use this key to manage the i- α pli version. Only ASCII characters may be used.

PackageURL: Mandatory key

i-appli (JAR file) URL (max. 255 bytes). If a relative path is used, the base directory becomes the ADF location. Only ASCII characters may be used.

AppSize: Mandatory key

An integer value expressing the JAR file size in bytes. In order to limit the JAR file size, this value will be no higher than 30720. Note that with DoJa-1.0 Profile, the value for the AppSize is limited to 10240 maximum.

AppClass: Mandatory key

Main class name used for launching the i-appli (w/full package name/max. 255 bytes). Only ASCII characters may be used.

AppParam: Optional key

Launch parameters for main class. Separate each parameter using spaces (max. 255 bytes). To obtain these parameters from the application program, use the `com.nttdocomo.ui.IApplication.getArgs()` method. Only ASCII characters may be used. If this entry is omitted, the application is launched without any parameters.

ConfigurationVer(KvmVer): Optional key

J2SE Configuration version. No entry signifies that the i-appli will work on all versions of the CLDC virtual machine. Although ConfigurationVer and KvmVer both have the same meaning, it is not possible to specify a single ADF for both of these at the same time. KvmVer is made available in order to guarantee compatibility with DoJa-1.0 Profile, and the setting values for this key must conform with the following format.

CLDC-x.x (x being a number from 0 – 9. The current version supports 1.0)

ProfileVer: Optional key (new with DoJa-2.0 Profile)

Profile version for i-appli Runtime Environment. If this key is not specified, it will indicate that i-applis will operate in all versions of the i-appli Runtime Environment. Setting values for this key must conform with the following format.

DoJa-x.x (where x is a value from 0 – 9.)

The below are supported in this version.

- DoJa-1.0 (supports the 503 series)
- DoJa-2.0 (supports the 504 series)
- DoJa-2.1 (supports FOMA02 models)
- DoJa-2.2 (supports FOMA03 spring models)
- DoJa-3.0 (supports the 505 series)

SPsize: Optional key

An integer value expressing the ScratchPad size in bytes. See Section 3.2 for a discussion of the limitations on ScratchPad size. If omitted, it is assumed that no ScratchPad is to be used.

In DoJa-3.0 Profile, a maximum of 16 ScratchPads can be internally segment-managed. To segment the ScratchPad, the individual byte numbers, separated by commas (no spaces before or after the commas), are enumerated as the SPsize key value. When accessing a ScratchPad, the ScratchPad's number (0 to 15 max) is specified in the URL, but that number is equivalent to the size specified by this key.

(If "SPsize = 1024,2048" is entered, ScratchPad 0 will have a size of 1024 bytes, while ScratchPad 1 will have a size of 2048 bytes.)

When segment-managing the ScratchPad, the sum of the sizes of the individual segments should stay within the size limitation.

LastModified: Mandatory key

Last time the i-appli was modified (JST). i-appli upgrades are conducted based on the contents of this key. The format is as follows.

Dow, DD Mon YYYY HH:MM:SS TimeZone

day of week, day, month, year, hours (24-hour clock), minutes, seconds, TimeZone

Omission of the time zone is possible. Since specification of a time zone was not supported with DoJa-1.0 Profile, it will be necessary to omit the time zone when sharing ADF between a mobile phone compatible with DoJa-1.0 Profile and an mobile phone compatible with DoJa-2.0 Profile (The time zone is not set in the ADF created by the emulator).

Both “GMT” and “JST” are applicable as time zones. Furthermore, it is also possible to set a time difference of “+0900” as a time zone. If no time zone is specified, “JST” will be adopted as default.

Sample entry: Fri, 25 Feb 2000 12:04:25 JST

UseNetwork: Optional key

Specifies communication method when using network functions. In the i-appli Runtime Environment, “http” is the only valid value (even when HTTPS is used, specify “http”). If this entry is omitted, the i-appli will not be allowed to make network connections.

TargetDevice: Optional key

Names of target mobile phone devices, separated by commas (max. 128 bytes). Do not specify this entry if your application targets all devices. Values are set for this key using one of the following formats.

- Individual specification of target model names. Model names are the same as those included in the user agent, and in situations where multiple target models exist, these names are to be separated by commas.

Example: X505i,Y505i,Z505i

- Group specification of all models in the target series. Specifically, the series name is added to the end of the keyword “all” to give a specification such as “all505.” In situations where multiple target series exist, these names are to be separated by commas. Please be aware that this specification method will not be correctly interpreted on mobile phones which support only DoJa-1.0 Profile.

LaunchAt: Optional key

If the i-appli is to launch automatically, use this entry to specify the time for it to launch. The first character is a format identifier. The i-appli Runtime Environment supports the “I” format identifier, which stands for “interval.”

- ✧ Time interval specification format (format identifier: I)

Format: I h

(where h is a one- or two-digit number representing hours, One space is needed between the “I” and the number.)

Sample specification: I 6 (launches every day at 6-hour intervals)

The interval is calculated from the time the i-appli is downloaded to the mobile phone.

If another application is already running, or the mobile phone is communicating, when the time comes for the i-appli to automatically launch, then the i-appli is unable to launch. If this happens, the mobile phone will notify the user that an i-appli failed to launch automatically. How this is done is up to the device manufacturer.

Depending on the manufacturer, there may be features in the mobile phone that allows the user to save settings so that i-appli will automatically launch at specific times. In cases where such a feature performs an automatic launch, the i-appli launch mode will not be a timer launch (IApplication.LAUNCHED_FROM_TIMER), but a normal launch (IApplication.LAUNCHED_FROM_MENU).

MyConcierge: Optional key (new with DoJa-2.0 Profile)

Declaration of a stand-by application. If the value is set as “Yes”, then it will indicate that the i-appli in question may be used as a stand-by application. i-applis without this specification will not be recorded as stand-by applications in the mobile phone.

UseTelephone: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which can use voice-call functionality. If the value is set as “call”, the corresponding i-appli will be able to make voice calls (using the `com.nttdocomo.util.Phone.call()`). i-applis without this specification will not be able to make voice calls.

UseBrowser: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which can use browser functionality. If the value is set as “launch”, the corresponding i-appli will be able to use the `com.nttdocomo.ui.IApplication.launch()` method to launch a browser. i-applis without this specification will not be able to launch browsers. Furthermore, i-applis for which stand-by launching was carried out will not be able to launch browsers, regardless of this key's setting.

LaunchByMail: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which can be launched from mail. The value specifies in ASCII format a single source mail address for which launching is permitted (50 bytes maximum). If this key's value is fully included in the source mail address as the result of rearward-match comparison, permission will be granted for launching of the i-appli from the corresponding mail. Note that if the key's value is specified as “any”, it will indicate that no restrictions are placed on source mail addresses.

LaunchByBrowser: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which can be launched from a browser. The value specifies in ASCII format a single web-page URL for which launching is permitted (255 bytes maximum). If this key's value is fully included in the URL being displayed on the browser as the result of forward-match comparison, permission will be granted for launching of the i-appli from the corresponding web page. Note that if the key's value is specified as “any”, it will indicate that no restrictions are placed on URLs.

AllowPushBy: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which has permission to be launched from an external interface. There are two types of mode of being launched from an external interface: launched as a result of reception of a vTrigger object from the infra-red port or launched from a two-dimensional barcode used in the code recognition functionality (optional, but not supported by some manufacturers).

<Launch as a result of reception of a vTrigger object from the infra-red port>

The `AllowPushBy` key value specifies in ASCII format a single format parameter as shown below (Maximum of 255 bytes, including the “Irda:” prefix).

Irda:<command>

The parameter <command> specifies the launch command as indicated in the vTrigger object for which i-appli launch is permitted. If this key's value is completely matched with the launch command from the vTrigger object requesting i-appli launching, then i-appli launch as a result of that vTrigger object will be permitted.

No blank spaces or tabs can be contained in the <command>.

<Launch from a two-dimensional barcode used in the code recognition functionality>

The `AllowPushBy` key value specifies in ASCII format a single format parameter as shown below (Maximum of 255 bytes, including the “Code:” prefix).

Code:<command>

The parameter <command> specifies the launch command as indicated in the two-dimensional barcode for which i-appli launch is permitted. If this key's value is completely matched with the launch command from the two-dimensional barcode requesting i-appli launching, then i-appli launch as a result of that two-dimensional barcode will be permitted.

No blank spaces or tabs can be contained in the <command>.

AppTrace: Optional key (new with DoJa-2.0 Profile)

Key for providing simple debugging functionality on the mobile phone during the i-appli development phase. If this key is specified as “on”, then the following functions will be activated on the mobile phone.

- When an abnormal end occurs for an i-appli as a result on an non-caught exception, the actual type of exception leading to this situation will be displayed.
- It will be possible to lock the character strings which are output from an i-appli to the standard output or standard error output, and these may be displayed after the i-appli has ended. However, the length of the character string which may be locked is limited (to 512 bytes maximum), and display within the limit range will be carried out sequentially from the item most recently written. Note that results calling `java.lang.Throwable.printStackTrace()` will not be recorded in this log.
- The functionality for display of JAM i-appli details will enter detailed mode.

When development has been completed for an i-appli, please ensure that this key is deleted from the ADF.

DrawArea: Optional key (new with DoJa-2.0 Profile)

Specifies the screen size adopted as a pre-condition by the i-appli (i.e., the i-appli's rendering area size). If an i-appli which requires a small rendering area is executed on a mobile phone with a large rendering area, then the i-appli will be executed in a centered, rectangular area of the specified size. It is not possible to carry out rendering from an i-appli in the space outside the rectangle area. (The color displayed in this space will depend on the manufacturer in question.)

Key values are specified as shown below.

`<width>x<height>`

(`<width>` indicates pixels in the lateral direction; `<height>`, in the vertical direction.

Only a lower-case character is valid for x)

Example: `DrawArea = 120x120`

The smallest value which can be set for `<width>` is 96, and the smallest value which can be set for `<height>` is 72. Furthermore, the rendering area size as specified for this key must be no larger than the rendering area size for the mobile phone attempting to download the i-appli. (Either the width or the height must be equal to, or smaller than, the size of the mobile phone attempting download.) If settings which do not satisfy these conditions are made, it will not be possible for that application to carry out downloading.

Note that if this key is not specified, the i-appli will be executed using the mobile phone's complete rendering area.

GetSysInfo: Optional key (new with DoJa-2.0 Profile)

Declaration of an i-appli which can refer the mobile phone's icon data. If an i-appli for which this key has the value “yes” is downloaded, then the user will be asked whether or not the mobile phone is to permit the referencing of icon data from the i-appli. i-applis for which this key is not specified or for which user confirmation has not been acquired will not be able to reference icon data.

Note that the `PhoneSystem.getAttribute()` method is used to acquire the reception condition for mail and messages.

[DoJa-2.0] [DoJa-3.0]

In DoJa-2.0 Profile, the icon information that can be referenced is the mail/message received state. In DoJa-3.0 Profile, in addition to the mail/message received state, the states of the signal strength, battery level, and Manner mode setting can also be referenced.

LaunchApp: Optional key (new to DoJa-3.0 Profile)

Declares that the i-appli will launch another i-appli (link mode or launcher mode) with application linking. If the key's value is set to “yes”, that i-appli can use the `com.nttdocomo.ui.IApplication.launch()` method to launch another i-appli. If not specified, the i-appli cannot launch other i-appli.

LaunchByApp: Optional key (added in DoJa-3.0 Profile)

Controls behavior if a launch attempt is made from another i-appli with application linking. If the key's value is set to “deny”, that i-appli will not permit linked operation with launcher mode from another i-appli. In i-appli operation with application linking, the launched i-appli will be operated normally even if it is a stand-by application; stand-by operation will not occur. Stand-by applications that do not expect to be launched normally can use this key to forbid linked operation.

AccessUserInfo: Optional key (added in DoJa-3.0 Profile)

Declares that the i-appli will access the mobile-phone's user data area with application linking. If the key's value is set to "yes", that i-appli can perform the following user data operations:

- Add phonebook entries, phonebook groups
- Add bookmarks
- Add schedules
- Add, select and retrieve images

GetUtn: Optional key (added in DoJa-2.1 Profile)

Declares that this i-appli will reference the mobile-phone's individual identification information. If this key's value is set to "terminalid", the mobile-phone unit's identification information can be referenced; if it is set to "userid", the UIM card's identification information can be referenced. When referencing both the mobile-phone unit's identification information and the UIM card's identification information, separate these two with a comma (no space before or after the comma).

When an i-appli with this key specified is downloaded, the mobile-phone asks the user to confirm permission for the i-appli to access the individual identification information. If this key is not specified in the i-appli, or the user does not grant i-appli permission, individual identification information cannot be referenced. Also, UIM identification information cannot be retrieved from mobile-phones which are not equipped with a UIM card.

LetPreserve: Optional key (added in DoJa-3.0 Profile)

Declares that i-appli which launch immediately after download forbid being saved to the mobile-phone. If this key's value is set to "deny", the user cannot save that i-appli to the mobile-phone.

If this key is not set, after the i-appli has finished executing, the mobile phone will ask the user whether or not to save the i-appli on the mobile phone.

For details on i-appli that launch immediately after download, see Chapter 14.

Below is a sample ADF. As you can see below, only one entry is allowed per line in an ADF.

```
AppName = BankingDemo
ConfigurationVer = CLDC-1.0
ProfileVer = DoJa-3.0
AppClass = banking.BankingDemo
AppVer = 1.0
PackageURL = http://aserver.com:8080/BankingDemo.jar
AppSize = 17556
LastModified = Fri, 25 Feb 2000, 12:04:25
```

13.5.2 Creating the JAR File

JAR files are a standard Java archive feature that are designed to hold a set of classes and resource files comprising a Java application in a compressed format. In the JAR file, the size is decreased to roughly 40% to 50% as compared to the total size of the file(s) before compression (the compression rate will vary depending on the contents of the JAR file).

The i-appli is also distributed to the mobile phone packaged in a JAR file. This not only decreases the amount of space needed to save the i-appli on the mobile phone, but also shortens the i-appli's download time. The individual files packaged in the JAR file are unpacked as needed by the JAM.

13.6 Testing the i-appli

Test the i-appli with the Emulator and on an actual i-appli mobile phone.

At the initial stages of development, it is more efficient and cost-effective to test and debug the i-appli on the Emulator. The Emulator lets you debug the operation of the user interface, I/O, application logic, and so on, then fix the application program and check the results on the PC immediately.

But the Emulator is not capable of perfectly emulating the behavior of the hardware, for instance in the cases of program running speed or network bandwidth. Moreover, each device has its own particular memory capacity and other limitations. It is therefore crucial to conduct final testing on an actual i-appli device, in order to make sure that the i-appli behaves properly.

For details on using the Emulator, see the i-appli Developing Environment User's Guide

There are some additional limitations to the Emulator Environment in terms of testing the i-appli. The Emulator provides an environment to execute the i-appli. It does not emulate the i-appli management behavior (the JAM features) that is on the actual mobile phones. For example, i-appli update is a behavior that the Emulator does not emulate since the Emulator does not implement the JAM's i-appli management features. Other behaviors that cannot be tested are:

- LastModified, TargetDevice, and LaunchAt keys. The Emulator does not provide functionality to emulate the behaviors of using these keys.
- ConfigurationVer (KvmVer key) is not checked against the CLDC virtual machine in the Emulator.
- Since the emulator reads and executes i-appli JAR files present on the development computer's hard-disk (i.e., in the emulator's application directory), the values specified for PackageURL are not used. When actually distributing a mobile phone, the URL needs to be set correctly in PackageURL in order to download the JAR file.

To install an i-appli on an actual mobile phone for testing, you will need to place the JAR file, ADF, and HTML content for downloading the i-appli on a Web server connected to the Internet. See Chapter 14 for details on this procedure.

Notes:

- In an i-appli Development Environment compatible with DoJa-1.0 Profile, it is necessary for MIDI-format files to be used as MediaSound data. Accordingly, when developing i-applis which use MediaSound in such an environment, it will be necessary to replace MIDI-format files to MFi format files when testing on an actual mobile phone. Be aware that in some cases, you will have to edit (the filename extensions of) the URLs of resources in the i-appli's source code.

Note that both MFi format files and MIDI-format files may be used as MediaSound data in an i-appli Development Environment compatible with DoJa-2.0 Profile or later.

- If your i-appli uses a client-server architecture, you will have to place an appropriate sever-side application on the Web server used for testing.

Chapter 14

Distributing the i-appli

This chapter contains information on setting up the Web server to distribute i-applis.

14.1 Configuring the Web Server for Serving the i-appli

As described in the previous chapter, the i-Mode compatible i-appli consists of the ADF (.jam file) and JAR file (.jar file). To download an ADF that has been placed on a Web server, this file must be referred to by an i-Mode HTML OBJECT tag. The following example illustrates the tag definitions in the HTML file in order to download the i-appli:

```
<OBJECT declare id="application.declaration"
data="http://www.nttdocomo.co.jp/java/abc.jam" type="application/x-jam">
<PARAM name="Param1" value="i-mode">
<PARAM name="Param2" value="i-appli">
</OBJECT>
Click
<A ijam="#application.declaration" href="notapplicable.html">here</A>
to download.
```

Below are descriptions of the tags used to download the i-appli.

- A tag and ijam attribute

The A tag is used to refer to the OBJECT tag corresponding to the i-appli targeted for download. The download process begins when the user selects the A tag.

Set the A tag's ijam attribute to the name specified in the OBJECT tag's id attribute. ijam is a newly created attribute for i-appli, and can't be parsed by conventional i-Mode mobile phones, so you should also add an href attribute to the A tag. That way, the conventional i-Mode devices will parse the href attribute, while the i-appli devices will parse the ijam attribute. Furthermore, it is not possible for both the ijam attribute and the ista attribute (as used for launching i-applis from a browser) to be active at one time. If both of these properties are specified at the same time, mobile phones which are compatible with DoJa-2.0 Profile will normally interpret the ista attribute, and consequently, the ijam attribute will be ignored. (In the case of mobile phone which are compatible with DoJa-1.0 Profile, correct interpretation of the ista attribute is not possible, and therefore, the ijam attribute will normally be interpreted.)

In other words, ijam attributes in the A tag should always be paired with an href attribute.

[DoJa-2.0]

Browser launching of i-applis and the A tag's ista attribute have been newly added for DoJa-2.0 Profile. For more details on either of these, refer to Chapter 11.

- OBJECT tag

The OBJECT tag is used to refer to the ADF corresponding to the i-appli. Set the name in the OBJECT tag's id attribute to the name referred to from the A tag's ijam attribute (identical in HTML). In the data attribute, specify the URL indicating the location of the ADF. Set the type attribute to the content type of the data pointed to by the data attribute (in this case, ADF). The content type of ADF is "application/x-jam".

Note that the i-appli JAR file is referred to from the PackageURL key in the ADF.

It is possible to include descriptions of the following PARAM tag within the OBJECT tag.

- PARAM tag

When an i-appli is downloaded and used for the first time^(*), the PARAM tag is used to specify the parameters which can be acquired using the `IApplication.getParameter()` method.

Up to a maximum of sixteen PARAM tags may be included within a single OBJECT tag. Furthermore, the total of all of the name attribute lengths and value attribute lengths as specified for all PARAM tags within a single OBJECT tag is limited to a maximum of 255 bytes. In the case of both name and value attributes, the value SJIS can be used to indicate Japanese text.

- (*) In situations where downloading and first execution of an i-appli is carried out through application linking (i.e., launching from a browser, mailer, external device, or i-appli), the details of the PARAM tag as set here will not be passed to the i-appli. In situations where first execution is carried out as a result of application linking, the parameters as passed from the application will become valid, and these will be passed to the i-appli launched by link. For more details regarding application linking, refer to Chapter 11.

[DoJa-2.0]

Parameter specification using the PARAM tag and the `IApplication.getParameter()` method have been newly added for DoJa-2.0 Profile.

The i-appli mobile phone checks the content type of the ADF file by looking at the `type` attribute of the OBJECT tag. This means that you do not have to configure the content type corresponding to the ADF file extension on the Web server side.

[DoJa-2.0]

With DoJa-2.0 Profile, no restrictions apply to the ADF extension. In contrast, this extension was required to be “jam” in the case of DoJa-1.0 Profile. (In cases such as where the actual form of an ADF’s URL is a CGI or the like, the URL for that CGI has “jam” added to its end.)

14.2 Configuring the Web Server for Server-side i-appli

Some of the client-server applications may require the a specific Web server configuration in order to run, depending on the technology applying the server-side i-appli. For instance, if a Java servlet is used in the server-side application, it will be necessary to install and configure a Java servlet on the Web server side.

Refer to the documentation regarding the Web server you are using and the technology you are implementing in your server-side application, and configure your Web server appropriately.

14.3 Manufacturer-specific i-appli Downloads

i-applis using i-appli optional features or i-appli manufacturer extension capabilities or that have strict screen size requirements are specifically designed by the developer to only run on certain manufacturer’s mobile phones. This type of i-appli should be appropriately installed just on those target devices.

The i-appli mobile phone makes the following mechanisms available in order to appropriately control the download of the i-appli for each device.

1. The HTTP request from the mobile phone contains a `User-Agent` request header for identifying the mobile phone device type. The Web server side can use the content of the `User-Agent` header to direct that user to an HTML download page displaying content appropriate for that device type.
2. The mobile phone side can refer to the `TargetDevice` key in the ADF to determine whether it supports the download i-appli. The developer can set the `TargetDevice` key appropriately so that only intended device types download their i-appli.

14.4 Launching i-appli Immediately After Download

In DoJa-3.0 Profile, launching immediately after download is supported so that downloading and launching an i-appli can be performed with a single click from the user. Because i-applis that launch immediately after download require no user confirmation, they are limited to using functionality for which basic user confirmation is necessary.

i-applis that launch immediately after download cannot call the API related to the below functions. Calling these API will cause an exception.

- Functionality which is permitted by the ADF `GetSysInfo` key (icon information references)
- Functionality which is permitted by the ADF's `GetUTN` key (individual identification information references)
- Functionality which is permitted by the ADF `AccessUserInfo` key (access to the user data area)
- Functionality which is permitted by the ADF `LaunchApp` key (i-appli linked operation)
- Use of ScratchPad
- Linked operation of i-appli update functionality (JAM) from i-appli.

To distribute an i-appli as an i-appli that launches immediately after download, the `ilet` attribute should be specified in place of the `ijam` attribute in the i-appli download link (A tag). Like the `ijam` attribute, it is necessary to use the `ilet` attribute together with the `href` attribute.

Below is an example HTML file for distributing an i-appli as an i-appli that launches immediately after download.

```
<OBJECT declare id="application.declaration"
data="http://www.nttdocomo.co.jp/java/abc.jam" type="application/x-jam">
<PARAM name="Param1" value="i-mode">
<PARAM name="Param2" value="i-appli">
</OBJECT>
<A ilet="#application.declaration" href="notapplicable.html">Click here</A>
to download and launch the i-appli.
```

In i-appli that launch immediately after download too, parameters that use the `PARAM` tag can be delivered and retrieved.

Notes:

- i-applis launch immediately after download is located in the equivalent position as i-applis launch from the browser implemented during application linking. Thus, in i-applis that launch immediately after download, the ADF's `LaunchByBrowser` key must be properly set to enable i-appli linked launch from an i-appli download HTML. For details on the ADF's `LaunchByBrowser` key, see Section 13.5.1.
- In i-applis that launch immediately after download, originally, after the i-appli download, the i-appli is limited to using functionality that can be confirmed or cancelled by the user, but HTTP(S) communications can be used in exception to this. In i-applis that launch immediately after download, HTTP(S) communication permission confirmation can be performed not at download time, but when the i-appli first attempts to begin HTTP(S) communication. When permission confirmation occurs, i-appli is interrupted.
- i-applis that are launched immediately after download are launched normally, not in stand-by, even if they are stand-by applications.
- After an i-appli that launches immediately after download normally finishes execution, the mobile-phone will ask the user whether or not to save the i-appli. If save is chosen, the i-appli will be saved on the mobile-phone as a normal i-appli, and after this user confirmation for HTTP(S) communication permission settings, etc., will be performed in accordance to the details of the ADF (at this time, the functionality usage limitations mentioned earlier in this section will be removed).

The developer can use the ADF `IletPreserve` key to keep i-appli that launch immediately after download from being saved to the mobile phone. See Chapter 13 regarding the ADF `IletPreserve` key.

Note that if an i-appli that has launched immediately after download is terminated by launching a browser by linked launch, the i-appli will not be saved.

- In an i-appli that launches immediately after download, the launch type retrieved by the `IApplication.getLaunchType()` method will be `IApplication.LAUNCHED_AS_ILET`. Further, if the i-appli is saved to the mobile phone after terminating, when next launched normally for the first time (launched by a user operation from the i-appli menu), the launch type will be `IApplication.LAUNCHED_AFTER_DOWNLOAD`.
- Trusted i-applis cannot be downloaded as i-applis which launch immediately after download.

Appendix A

FOMA Mobile Phone i-αpli Runtime Environment—FOMA03 Latest Specification Models

Below are special notes for the i-αpli Runtime Environment installed in the FOMA mobile phones planned to be on the market from winter 2003 (hereinafter referred to as FOMA03 latest specification models).

Mobile phones of FOMA03 latest specification models are provided with the i-αpli Runtime Environment conforming to the DoJa-3.5 Profile. The DoJa-3.5 Profile includes some functions extended from the DoJa-3.0 Profile. As the wireless network capacity for FOMA services and the service contents have been extended, optimization has been made to some specifications of the i-αpli Runtime Environment for FOMA mobile phones.

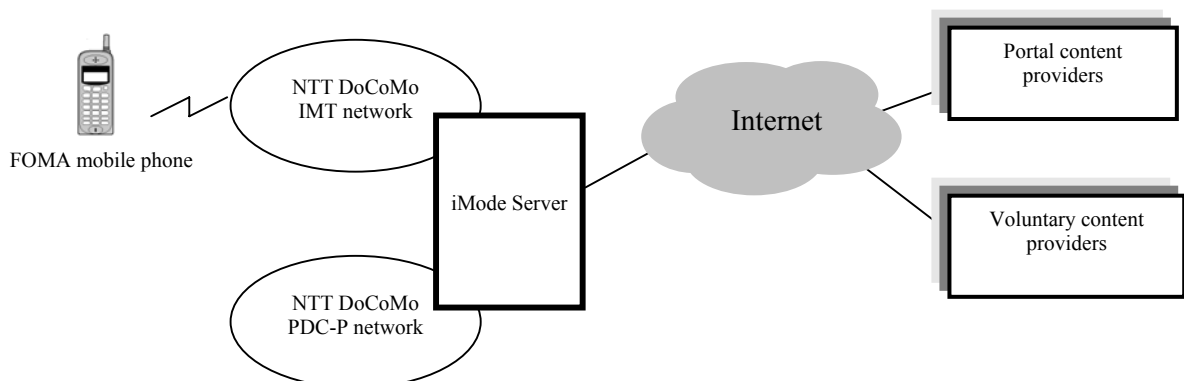
For the FOMA03 latest specification models, it is possible to produce i-αpli showing great power of expression due to increased JAR file size and ScratchPad size, enhanced drawing functionality, and migration of some i-αpli optional APIs to i-αpli standard APIs.

A.1 i-αpli Runtime Environment Optimization for FOMA

The following describes the DoJa-3.0 Profile items optimized for FOMA mobile telephones.

A.1.1 Network Architecture

FOMA services use W-CDMA wireless network (hereinafter referred to as the IMT network) employed for the global standard IMT-2000 (International Mobile Telecommunication 2000) of the next-generation mobile telecommunication system. Application download and HTTP communications are implemented via this IMT network.



The IMT network allows high speed packet communications by mobile phones up to 64 Kbps data send and up to 384 Kbps data receive (best effort type, respectively), enabling high speed communication processing from applications.

While the conventional i-mode compatible mobile phones are implemented on the basis of HTTP/1.0, the FOMA mobile phones are implemented on the basis of HTTP/1.1. This also applies to HTTP communications from i-applis. The FOMA mobile phones cannot use the HEAD request method specified for HTTP communications. Only two types of request methods are available for the FOMA mobile phones: GET and POST.

A.1.2 Application Size and Network Communication Size

On FOMA03 latest specification models, the application size (JAR file size and ScratchPad size) is extended as shown below.

	FOMA03 Latest Spec. Models	Reference: PDC (505 Series)
JAR file size	100 Kbytes	30 Kbytes
ScratchPad size	400 Kbytes (*1)	200 Kbytes

*1: The rules for ScratchPad division management is still the same as the conventional (up to 16 divisions available).

ScratchPad access is high cost processing for mobile phones which have restrictions in performance and resources. For i-applis using a large ScratchPad area, properly specify the `pos` and `length` options for the URL during ScratchPad access so that only the necessary area will be accessed using minimum memory.

For the FOMA03 latest specification models, the network (HTTP(S)) communication size from i-applis is extended as shown below.

	FOMA03 Latest Spec. Models	Reference: PDC (505 Series)
Uplink communication size	80 Kbytes	10 Kbytes
Downlink communication size	150 Kbytes	20 Kbytes

A.1.3 Media Data

The DoJa-3.0 Profile supports the following types of media data.

- Still image: JPEG data, GIF data
- Moving image: GIF data (animation GIF)
- Sound: MFi data

In addition to the above types of data, the data types shown below are also available on FOMA mobile phones.

(1) Sound: SMF (Standard MIDI File) data

A `MediaSound` object is generated in the same way as MFi data and is played back using `AudioPresenter`. FOMA services use the SMF format as the standard sound data format. However, note that MFi may not be supported depending upon mobile phone models to be released in the future.

The DoJa-3.0 Profile supports the function for playing back different sounds at the same time using more than one `AudioPresenter` object. However, availability of simultaneous playback of SMF and MFi data varies with the mobile phone manufacturer. In order to simultaneously play back multiple sounds, it is necessary to unify the sound data format to either SMF or MFi.

(2) Moving image: iMotion data

A `MediaImage` object is generated in the same way as animation GIF data and played back using `VisualPresenter`. A `MediaImage` object generated from iMotion data can only be used for playback with `VisualPresenter`. For example, when an attempt has been made to extract an `Image` object from that `MediaImage` object using the `getImage()` method or that `MediaImage` object was saved or re-obtained using the `ImageStore` class, the operation varies with the mobile phone manufacturer.

In DoJa-3.5 Profile, some APIs of the concerned classes are extended to handle moving images of the iMotion format.

- **Specifying the player mode**

iMotion data can be played back in two modes; native player mode and inline player mode. In the native player mode, the current i-appli is temporarily suspended by a playback instruction (`VisualPresenter.play()`) and the native iMotion player is started. In the inline player mode, the `VisualPresenter` component arranged in `Panel` plays iMotion data without suspending the current i-appli. (The inline mode is optional and its availability varies with the mobile phone manufacturer.)

The player mode can be selected by setting the `PLAYER_MODE` attribute in the `setAttribute()` method of the `VisualPresenter` class.

- **Specifying the voice data playback mode**

iMotion data is media data which can involve voice. For iMotion data playback in the inline player mode, it is possible to select whether to play back voice data included in the iMotion data. In order not to play back voice data, media sounds can be simultaneously played back with `AudioPresenter` prepared separately from `VisualPresenter`.

The voice data playback mode can be selected by setting the `AUDIO_MODE` attribute in the `setAttribute()` method of the `VisualPresenter` class.

- **Compatibility with progressive download**

There are two types of iMotion data download: normal download and progressive download. The normal download mode allows iMotion data to be played back after download of all data is completed on the concerned mobile phone, whereas the progressive download mode allows iMotion playback to be started when a specified volume of data has been obtained and, subsequently, implements data download and playback concurrently. In DoJa-3.5 Profile, the `MediaManager` class is newly provided with the `getStreamingImage()` method to be compatible with progressive download on the API level. For a `MediaImage` object generated from iMotion contents compatible with the progressive download using that method, data download and playback are concurrently carried out upon reception of the playback instruction in the `VisualPresenter` class.

An iMotion file of up to 150 Kbytes can be played back from i-applis. It is not possible to play back iMotion data from a standby i-appli in the inactive mode. When an attempt is made to play back iMotion data from a standby i-appli in the inactive mode, the behavior varies with the mobile phone manufacturer.

A.1.4 Application Lifecycle

Some FOMA mobile phones can simultaneously launch multiple device applications (e.g., browser and voice-call functionality) for carrying out i-mode site browsing and voice calling at the same time. This section describes influences on the i-appli lifecycle of such mobile phones.

(1) Changeover from the currently executed i-appli to another application by the user operation

As a general rule, if the user has instructed (task key operation or such) to change to another application during execution of an i-appli, that i-appli is suspended. The suspended i-appli is resumed when the operation target is changed again from the application to that i-appli by the user operation. The applications that can be entered in the launched state at the same time as an i-appli vary with the mobile phone manufacturer. If the user instructs a mobile phone of the concerned model to launch an application program (e.g., browser), which cannot be launched concurrently with an i-appli, during launch of an i-appli, the i-appli terminates and the specified application is launched.

An application program can obtain the suspend reason of i-appli execution and an event caused during the suspend by using the `IApplication.getSuspendInfo()` method. For the events which occur during i-appli suspend, DoJa-3.5 Profile is additionally provided with `IApplication.SUSPEND_MULTITASK_APPLICATION` to distinguish the suspend caused by the application changeover described in this section.

(2) Action for an incoming mail during execution of an i-appli

PDC mobile phones are designed to inform the users of unreceived mail in the i-mode Center using a specified icon or such without receiving that mail if they receive an incoming mail signal during execution of an i-appli (during normal launch of an i-appli or in the active mode during execution of i-appli stand-by launching). In contrast, FOMA mobile phones receive mail in the background without suspending the current i-appli when they receive an incoming mail signal in such state. It may be necessary to consider this point as the message i-appli functionality, which is a trusted API, sends/receives data using mail as the medium. If an i-appli uses a functionality (such as the infrared remote control functionality), which cannot be executed concurrently with packet communications, during mail receive in the background, an exception occurs.

Even on FOMA mobile phones, mail is received by suspending an i-appli, which is in the inactive mode during execution of stand-by launching or in the sleep mode, in the same way as PDC mobile phones.

A.2 Additions/Changes to APIs in DoJa-3.5 Profile

Below is the description of additions and changes from DoJa-3.0 to DoJa-3.5 Profile APIs.

A.2.1 Transition from i-appli Optional APIs to i-appli Standard APIs

In DoJa-3.5 Profile, the functions shown below are migrated from i-appli optional API to i-appli standard API. For details on these functions, refer to the “i-appli Content Developer’s Guide for DoJa-3.0: i-appli Options and Extensions” and the “i-appli Content Developer’s Guide for DoJa-3.0: API Reference.”

(1) Code recognition functionality

The code recognition functionality has been migrated to i-appli standard API by including i-appli linked launch from two-dimensional barcodes. The code recognition functionality for the i-appli standard APIs is provided by the `com.nttdocomo.device.CodeReader` class.

(2) Image map

Image maps for i-appli standard APIs are provided by the following class and method.

- `com.nttdocomo.ui.ImageMap` class
- `com.nttdocomo.ui.Graphics.drawImageMap()` method

(3) Paletted image

Paletted images for i-appli standard APIs are provided by the following classes.

- `com.nttdocomo.ui.Palette` class
- `com.nttdocomo.ui.PalettedImage` class

Paletted images can be used for image drawing implemented by the `sprite`, `image map`, and `Graphics.drawImage()` method. Availability of paletted images for others varies with the mobile phone manufacturer.

(4) Sprite

Sprites for i-appli standard APIs are provided by the following classes and method.

- `com.nttdocomo.ui.Sprite` class
- `com.nttdocomo.ui.SpriteSet` class
- `com.nttdocomo.ui.Graphics.drawSpriteSet()` method

For the i-appli optional API `sprite`, the `setRenderMode()` method was provided to specify the raster operation during `sprite` drawing. However, it is not supported by the i-appli standard API `sprites`.

A.2.2 Newly Added APIs

Below are APIs newly added in DoJa-3.5 Profile. For details on their functionality, refer to the “i-appli Content Developer’s Guide for DoJa-3.0: API Reference.” This section does not cover any of i-appli option/extension APIs nor trusted APIs.

(1) `com.nttdocomo.ui.Graphics` Image drawing with affine transformation

The 3x3 affine transformation matrix as shown below is applied to the respective coordinates on an image to implement drawing onto a calculated coordinate group. In the equation below, x and y refer to the coordinates on the original image, x' and y' refer to the converted coordinates (coordinates subject to drawing), and m00 to m12 refer to the affine transformation matrix elements which can be specified from an application program, respectively.

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \frac{1}{4096} \begin{pmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ 0 & 0 & 4096 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Use of this functionality enables image drawing by adding such transformation as enlargement, reduction, and rotation to the image

(2) `com.nttdocomo.util.Base64` Base64 encoder/decoder

The Base64 encoder/decoder is a utility functionality provided by the `com.nttdocomo.util.Base64` class. It is possible to carry out Base64 encoding of a specified arbitrary byte string or character string and decoding of Base64 encoded data.

Glossary

API	Application Program Interface
Applet	An applet is a small program that can be sent along with a Web page to a user. The i-appli is not an applet.
Bps	Bits per second
CGI	Common Gateway Interface
CHTML	Compact HTML
CLDC	Connected Limited Device Configuration
HTTP	Hypertext Transfer Protocol
HTTPS	Secure Hypertext Transfer Protocol
IDE	Integrated Development Environment
J2EE	Java 2 Enterprise Edition
J2ME	Java 2 Micro Edition
J2SE	Java 2 SDK, Standard Edition
JAM	Java Application Manager
JAR	Acronym for Java Archive file
JDK	Java Development Kit
JNI	Java Native Interface
JVMS	Java Virtual Machine Specification
MIME	Multi-Purpose Internet Mail Extensions is an extension of the original Internet email protocol that lets people use the protocol to exchange different kinds of data files on the Internet.
Servlet	A servlet is a single, standard, generic mechanism for developing modular server-side Java code that generates dynamic Web content.
SSL	Secure Socket Layer
UI	User Interface
Unicode	A 16-bit encoding for representing characters
URL	Uniform Resource Locator is the address of a file (resource) accessible on the Internet.