Java TV™ API Technical Overview:

The Java TV API Whitepaper

Version 1.0
November 14, 2000

Authors: Bart Calder, Jon Courtney, Bill Foote, Linda Kyrnitszke, David Rivas, Chihiro Saito, James Van Loo, Tao Ye

Please send any comments on the *Java TV API Technical Overview* to javatv-comments@sun.com.

Contents

Introduction

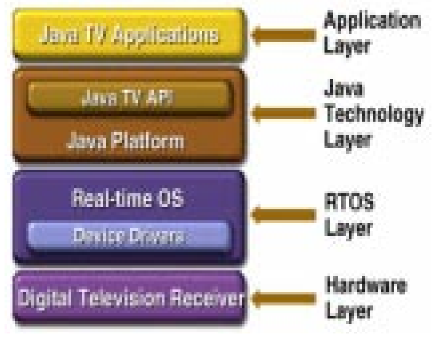
This document describes the Java TV™ API, an extension of the Java™ platform designed for developers who are producing Java-based interactive television content. The Java TV API gives programs written in the Java programming language control of broadcast television receivers and set-top boxes.

A key purpose of the Java TV API is to provide application developers with the ability to easily build applications that are independent of the underlying broadcast network technology on which they will be deployed. For example, many applications need basic information from a service information database, such as a list of the names of services currently available. The Java TV API provides an abstraction that permits applications to obtain this information in a manner independent of the service information protocol currently in use. This allows an application to be written and then reused in a variety of network environments. The Java TV API is designed throughout with a high level of abstraction from hardware and over-wire protocols.

Wherever possible, the Java TV API relies on an application environment to provide general purpose APIs. For example, file storage APIs and network communication APIs are provided by the application environment. In some cases, functionality that might be available on a set-top box is exposed with another Java extension. For example, in set-top boxes that provide telephone service, the JavaPhone™ API may be used.

FIGURE 1 shows the Java TV API and application environment as they are typically implemented on digital receivers. Programmers write applications to the Java TV API and application environment APIs, allowing their applications to be largely unaware of the underlying RTOS and hardware details.

FIGURE 1 A Typical Television Receiver Implementation



This document is a description of the API elements that comprise the Java TV API and is intended to be viewed along with the actual APIs in javadoc or other form. Because the Java TV API is designed to scale across a wide variety of possible implementations, this document does not describe minimal or performance-related hardware and software requirements.

1.1 Television Receivers

A television receiver may process either analog signals, digital signals, or both. Signals are usually broadcast to receivers over terrestrial, cable, or satellite networks. A digital signal permits a wider variety of content to be broadcast than does an analog signal. A digital broadcast might contain other types of information along with the digitized audio-video, such as a Java application. Television receivers come in a broad range of functionality and capabilities. The Java TV API provides access to the functionality that is commonly found on these receivers and scales across different receiver implementations.

While there are a wide variety of television receivers with different capabilities, it is useful to categorize receivers into three major types based on the kind of network connection a receiver supports: enhanced broadcast receivers, interactive broadcast receivers, and multi-network receivers. Each type of receiver builds on the capabilities of the previous type, as described below:

1. **Enhanced Broadcast Receivers**
   Enhanced broadcast receivers are capable of providing traditional broadcast television that is enhanced with graphics, images, text and can be controlled by a Java language-

based program provided as a part of the broadcast. Such receivers support local viewer interactions, including input from a remote control, on-screen graphical elements, selection among multiple audio-video streams, and switching among displays that augment audio-video presentation. Enhanced broadcast receivers receive data from a head-end or server, often carried via a broadcast file system. However, enhanced broadcast receivers have no return channel to the broadcaster, and therefore do not imply interaction with a head-end or server.

2. **Interactive Broadcast Receivers**
   Interactive broadcast receivers include a return channel to the broadcaster that provides communication with a head-end or server. Such receivers are capable of providing electronic commerce, video-on-demand, email, and local chat-style communication. Interactive broadcast receivers include the capabilities found on enhanced broadcast receivers.

3. **Multi-Network Receivers**
   Multi-network receivers provide access to more than a broadcast network and a return channel. Multi-network receivers include Internet capable receivers and receivers providing other communication services, such as local telephony. Such receivers can be home telecommunication hubs and provide diverse services, such as Internet browsing. Multi-network receivers include the capabilities found on both enhanced broadcast receivers and interactive broadcast receivers.

1.2 Television-Specific Applications

The Java TV API characterizes television programs as services. This is an abstraction that provides a common way to refer to a wide variety of content that may appear in a broadcast environment. For example, a service can refer to a regular TV program with its synchronized audio and video or to an enhanced television broadcast that contains audio, video, and a Java application that is synchronized with the broadcast. The Java TV API provides a means for selecting services, accessing a database containing service information, controlling a television-specific media player, and accessing data that is broadcast along with a television signal.

Content developers can write many types of television-specific applications, or services, including electronic program guides, program-specific applications, stand-alone applications, and advertisements. Some of the characteristics of these kinds of applications are described below.

1.2.1 Electronic Program Guides

Electronic program guides (EPGs) are some of the more common applications found on today's television receivers. An EPG's primary function is to provide the viewer with an overview of current and upcoming television programs. Usually, an EPG also changes the channel to the viewer’s selection. With this type of application, interactive performance and short start-up times are critical to a positive user experience.

1.2.2 Program-Specific Applications

Program-specific applications are created for and deployed along with specific service audio-video programming. Examples include an application deployed with a game show that allows viewers to play along at home, and an application that provides interactive information about a sporting event. These applications have several key requirements. For instance, the receiver hosting the application must be able to suspend the application when the viewer changes the channel.

1.2.3 Stand-alone Applications

Stand-alone applications appear to run unattached to normal television programming. An example is a stock ticker application that obtains data from a secondary network, displaying prices on screen. The user may be able to lock this application to the screen so that it remains as they change to another channel.

1.2.4 Advertisements

Advertisement applications are applications that augment the audio/visual content of a commercial. Such applications typically run for only the duration of the commercial, therefore, they are extremely short-lived. The actual downloading of the advertisement application may actually take place before the commercial starts. The application is typically stopped and discarded at the end of the commercial.

1.3 Features of the Java TV API

The Java TV API is a programming interface targeted at developers of interactive television services and other types of software applications that run on digital broadcast receivers. The major capabilities provided by the Java TV API for various types of applications are described below.

- **Accessing Services and Service Information**
  The Java TV API represents television programming, both traditional and interactive, as a set of individual services. The service information APIs provide support for obtaining service information (SI), which can be used to select a service.

  The SI database provides applications access to information about what services are available during runtime. Access to the SI database is through the SI manager. If an application is not interested in every service available, the SI manager permits filtering operations to find services of interest. The views of the SI database that have been defined are for controlling navigation, EPGs, and MPEG-2 delivery. For more information, see Chapter 3, “Services and Service Information”.

  Selecting Services
  Service selection APIs are used to select a service for presentation. The mechanics of selection are determined by the components of the service, and include starting an application if an application is a part of the service. For more information, see Chapter 4, “Service Selection”.

- **Controlling the Broadcast Pipeline**
  The Java TV API uses the Java^{TM} Media Framework (JMF) to represent the broadcast pipeline of a receiver. The JMF defines sources of data and handlers of content. The Java TV API makes a similar distinction for a broadcast pipeline. A tuner-demultiplexer-conditional access (C/A) subsystem is the source of data in the JMF sense, while the decoder-framebuffer-audio output is the content handler; see FIGURE 2. For more information, see Chapter 5, “JMF and the Broadcast Pipeline”.

- **Accessing Broadcast Data**
  A service is modeled as a multiplex of analog and digital data streams. In many cases these streams are not directly available to an application (e.g., audio/video streams). However, the multiplex may have streams of digital data that are available to an

application. The broadcast data APIs provide support for access to broadcast file systems, streaming data and encapsulated IP data. For more information, see Chapter 6, “Broadcast Data APIs”.

- **Managing Application Lifecycle**
  The sequence of steps by which an application is initialized, undergoes various state changes and is eventually destroyed is collectively known as the application lifecycle. The Java TV API defines a lifecycle for applications that run on digital broadcast receivers.

  Such applications are called Xlet applications. An Xlet is either resident on the receiver or can be downloaded and controlled by an application manager, which is part of a digital television receiver's software operating environment. The application manager manages an Xlet's lifecycle state changes. Each receiver has a resident application manager capable of providing an Xlet access to its environment through an application context passed to the Xlet during its initialization. For more information, see Chapter 7, “Application Lifecycle”.

Environment

The following sections, Hardware Environment and Software Environment, provide a model describing features provided by the environment in which the Java TV applications run. This model is not intended to be inclusive of all aspects of a television receiver, but simply to describe the pieces of the receiver to which the Java TV API and Java application environment provide access.

2.1 Hardware Environment

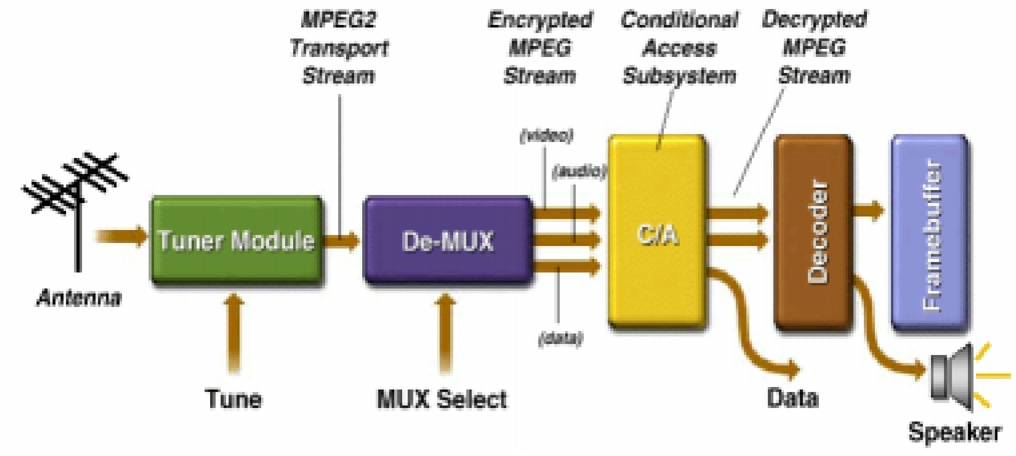
This section describes elements that make up a broadcast receiver. It is intended to be explanatory and provide context and definitions for concepts that are part of the Java TV API. This section is not intended to specify hardware requirements for the API.
The television receiver gets video, audio, and data from the broadcast stream and processes them through a broadcast media and data pipeline. The receiver gets the media and data in specific formats, called protocols, and decodes them using a variety of decoders specific to these protocols.

A distinguishing characteristic of a television receiver, relative to typical computing devices, is that the receiver is designed around a broadcast media pipeline. The broadcast media pipeline typically consists of a set of subsystems, such as a digital tuner, a demultiplexer, a conditional access module, a collection of media decoders and a rendering subsystem, through which the media flow. The Java TV API does not require that all subsystems be present. For instance, a receiver may have no conditional access subsystem or may not have a digital tuner. The Java TV API provides an abstraction that allows the application programmer to remain unaware of the details of the underlying hardware environment. However, the Java TV API assumes that the broadcast receiver has some sort of broadcast pipeline.

To illustrate, the following are the elements of a typical pipeline and the steps taken as an RF signal passes through it and is processed in a digital broadcast receiver (see FIGURE 2.) (This diagram is an example of one particular pipeline that could exist. Many other pipeline configurations are possible.)

1. An RF signal is tuned.
2. The tuned RF signal is demodulated into a digital signal, carrying an MPEG-2 transport stream.
3. The transport stream is passed through a demultiplexer and broken into multiple streams (e.g., audio, video, and data).
4. The video and audio streams are fed through a conditional access (C/A) subsystem, which determines access privileges and may decrypt data.
5. The decrypted audio and video streams are fed to a decoder, which converts them into signals appropriate for the video and audio output devices.

FIGURE 2 **Typical Enhanced Broadcast Digital TV Broadcast Pipeline**



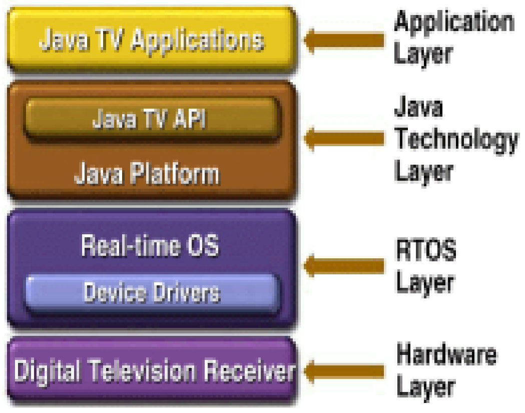
2.2 Software Environment

The software environment on a digital receiver typically consists of a Java application environment, the Java TV API, and supporting applications. In addition, the software environment typically includes a Real-Time Operating System (RTOS).

As shown in FIGURE 3, at the highest layer of the software environment, the Application Layer, an application can use the Java TV API and the Java packages from the layer below, the Java Technology Layer. Java applications execute in the application environment's virtual machine (VM). The Java TV API abstracts the functionality exposed by the lower-level libraries to control the hardware operations of the receiver.

The RTOS provides the system-level support needed to implement the Java technology layer. In addition, the RTOS and related device-specific libraries control the receiver through a collection of device drivers.

FIGURE 3 **Typical Software Stack on a Digital TV Receiver**



2.3 Application Environment

Applications designed to run on a broadcast receiver may take advantage of the application environment APIs, as well as the features built into the Java VM. This section describes the major aspects of broadcast receiver capabilities that are provided by the application environment and VM, apart from the Java TV API.

The APIs of a Java application environment are organized into functional groups called “packages”. The PersonalJava^{TM} application environment is typically used for devices with constrained memory footprints, such as television receivers. The PersonalJava application environment specification includes several useful packages:

- **I/O**
  The `java.io` package provides data input/output facilities using the classes `java.io.InputStream` and `java.io.OutputStream` and their subclasses.

- **Networking**
  The `java.net` package provides access to network functions using such classes as `java.net.URL`, `java.net.InetAddress`, and `java.net.Socket`.

- **Graphics toolkit**
  The `java.awt` package provides graphics rendering and window services to applications using such classes as `java.awt.Canvas`, `java.awt.Font`, and `java.awt.Scrollbar`.

- **System functions**
  Classes such as `java.lang.Thread` and `java.util.EventObject` provide applications with system-level functionality. The application environment also includes useful utility classes, such as `java.util.Hashtable` and `java.util.Calendar`.

2.3.1 Storage and Input/Output

The application environment package `java.io` provides abstractions for stream-based I/O, file-based I/O, and a wide variety of buffering options. Flash ROM systems, local hard drives, and server-based remote storage systems can all be accessed with `java.io`. Some environments may support the use of a system error stream for providing service operators with information on the status of receivers and their usage; such streams also use `java.io`.

Additionally, broadcast data streams and file systems require the use of `java.io`. For more information, see the section on broadcast data APIs.

2.3.2 Return Channel and Non-Broadcast Network Access

The package `java.net` provides an environment for accessing network sockets and HTTP connections, and for parsing URLs. In conjunction with `java.io`, this package provides the required functionality to access and manage an IP return channel or to access IP data encapsulated in MPEG transport streams.

2.3.3 Security

The application environment provides the foundation upon which network operators and standards organizations can define their own security models and policies. The Java TV API does not dictate a particular security model or policy, but uses the JDK 1.2 security architecture to express the security policies that are provided by the application environment.

This solution gives network operators and standards organizations the freedom to redefine their security models as future needs change. It also allows broadcasters to incorporate legacy security mechanisms into the platform. The remainder of this section describes some important security concerns associated with interactive television and the API support provided for each.

- **Conditional Access**
  The conditional access (C/A) subsystem controls the management of a set of authentication keys used to descramble or decrypt downstream video or data streams. The Java TV API does not define a mechanism for acquiring or managing C/A keys or decryption algorithms. There are a wide variety of deployed systems and a few standard interfaces, including APIs, defining C/A architectures.

  Rather than provide a high-level C/A subsystem API, the Java TV APIs express C/A subsystem interaction through the service selection APIs and the SI database. For more details, see Chapter 3, “Services and Service Information” and Chapter 4, “Service Selection”.

- **Secure Communication**
  Secure communication is important for protecting confidential information, such as financial data or electronic mail. Secure bi-directional TCP/IP connections can be achieved using SSL (secure sockets layer) and TLS (Transport Level Security) connections. The Java Secure Socket Extension (JSSE) is a Java standard extension for making SSL and TLS connections. The JSSE includes classes in the `javax.net` and `javax.net.ssl` packages that enable applications to access secure communications in a way that builds on the services present in `java.io` and `java.net`.

- **Virtual Machine**
  The Java VM is designed to provide secure execution of code. Bytecode verification insures the validity of the instructions that the VM executes. Class loading mechanisms

protect how code is loaded into the machine and can guarantee the validity of the code's source. The absence of direct memory-pointer manipulations from the Java language eliminates the risk of corruption due to code masquerading as data or stack-overflow based attacks. These techniques combine to provide a uniquely safe execution environment and augment the other security mechanisms.

2.3.4 Abstract Window Toolkit

The Java Abstract Window Toolkit (AWT) provides a collection of basic tools with which to build user interface (UI) components, or widgets. AWT also provides a large collection of native widgets. Vendors, consortia, and standards can define widgets to reside on receivers. Applications may also use the AWT to bundle special-purpose widgets supporting a particular look and feel. Such application-specific widgets are usually downloaded with the application.

Services and Service Information

A service is a collection of content for presentation on a receiver. This collection is handled as a unit within the Java TV API. Services can be selected for presentation. Television viewers often refer to this concept as a “television channel.” On today’s advanced television receivers, a service might not just consist of a single audio and video stream, it may consist of multiple audio and video streams as well as data.

Services have characteristic service information (SI), which is stored in the SI database. SI describes the layout and content of an audio/video/data stream, such as the MPEG-2 transport stream.

The Java TV API uses Locator objects to reference SI elements. A given locator may represent a network-independent object and have multiple mappings to network-dependent locators. The Java TV API provides methods for discovery of such circumstances and for transformation to network-dependent locators.

Various protocols for transmitting SI are used and standardized today. For example, DVB-SI is used in various satellite, cable, and terrestrial systems; the ATSC A56 standard is used on both satellite and cable; and the new ATSC PSIP (A65) is used on terrestrial and cable DTV. There are also a wide variety of private protocols. The Java TV API provides an abstraction of SI protocols, therefore a Java TV application does not have to be aware of the SI protocol that delivers information to the receiver. As a result, the application is not required to have special code to run in various environments, such as DVB-based, SCTE-based or ATSC-based systems.

3.1 Services and Service Information Definitions

- **service** - a collection of service components intended to be provided together as defined by the content provider. Each service has service information (SI) associated with it, and every service must have its SI made available to the receiver. An SI entry is one of the defining properties of a service.
- **service information (SI)** - information describing the content of a service or services. This includes basic information to present the components of the service as a coherent whole, as well as meta information such as the maturity rating of the service.
- **service component** - a "mono-media" element such as a video stream, an audio stream, a Java application, or some other data type that can be presented without needing extra information. A service will contain one or more service components, and a service component may be shared by more than one service.
- **service locator** - the information about a service needed to resolve it into a physical address for presentation.
- **SI database** - a database that stores service information accessible by television applications.
- **SI manager** - the primary access point to the underlying SI database. The SI manager reports changes related to the available SI elements and is capable of resolving a service locator into the meta data associated with the service.
- **SI element** - an object that represents a piece of service information.

3.2 SI Packages

The SI database object model allows various views of SI, based on an application's needs. Each view is represented as a package in the Java TV SI APIs. The SI API packages are Service, Navigation, Guide, and Transport.

- **service package: javax.tv.service**
  The service package provides the primary point of access to the SI database and contains classes common to the other SI packages, such as the Service and SIElement interfaces.

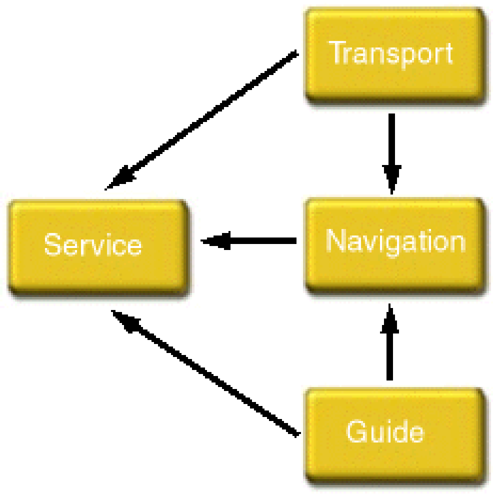
- **navigation package: javax.tv.service.navigation**
  The navigation package contains classes that are used to navigate the existing services (which are called Services in DVB and Virtual Channels in ATSC).

- **guide package: javax.tv.service.guide**
  The guide package contains classes useful for electronic program guides (EPGs), including program schedules, individual program events, and program ratings.

- **transport package: javax.tv.service.transport**
  The transport package represents the MPEG-2 delivery mechanism.

FIGURE 4 depicts the SI API packages. The arrows in the diagram indicate the dependencies among the packages.

FIGURE 4 SI API Package Dependencies



Most digital TV receivers will be unable to cache all the SI data in memory. The receiver will cache a subset of the SI data, consisting of the most useful information, but when it needs to retrieve data not stored in memory, the receiver will parse the transport stream. Because access to the transport stream may take a significant amount of time, the SI APIs provide asynchronous access to information that is not cached.

The SI APIs also provide a flexible mechanism for future extension of the API through the extension of SIElement, which would allow access to additional information.

3.2.1 Service Package

The service package provides several features used by the other SI packages:

- Base classes extended by the other SI packages.
- The event notification mechanisms for SI element changes detected in the transport stream and events delivering asynchronous requests.
- Exceptions.

Because most of the SI elements are interfaces rather than classes, an application does not have a way to directly instantiate an object that implements the specified interface.

Applications can register with the various objects to be notified when SI elements change in the broadcast. The SIChangeListener object and the SIChangeEvent object support the standard Java event model. The types of objects that support change notification are:

- The ServiceDetails object, which reports changes related to its associated ServiceComponent objects.
- The Transport object, which reports changes to carried services and network-definition related tables represented by the TransportStream object, the Network object, and the Bouquet object.
- The ProgramSchedule object, which reports changes detected in any one of the ProgramEvent objects in the schedule.

The service package also provides a mechanism to deliver data asynchronously. This functionality is provided by the SIRequestor, and SIRequest interfaces. A caller of an asynchronous method registers as an SIRequestor in order to later receive a callback once the requested data is available. The SIRequestor object receives the requested data or an indication of a failure. The SIRequest object is provided to cancel asynchronous requests if they are no longer needed by the caller.

3.2.2 Navigation Package

The navigation package provides two types of functionality:

- Mechanisms to request more detailed information about services and their service components.

- A mechanism to group Service objects into collections based on various grouping criteria.

The main navigation function is represented by several objects. The SI manager is the primary access point to the underlying SI database. It can generate a collection of Service objects, called a ServiceList, based on the selection criteria represented by the ServiceFilter object. The collection can then be used to sort, either by channel numbers or by channel names, and navigate through the Service objects. The base class ServiceFilter can be used to generate the default collection (using no filtering criteria), which represents all services installed on the receiver.

The Service object itself contains only the minimal information (such as locator, channel name and number) needed for navigation. Additional information about the service is contained in the ServiceDetails object. The ServiceDetails object provides information related to conditional access, the delivery mechanism of the service, and the time when the information about the service was last updated.

A set of channel components is associated with a Service object. It can also be associated with a specific ProgramEvent object, if such information is available. The current ProgramEvent object provides the same components as the Service object carrying the current program.

3.2.3 Guide Package

The guide package includes functionality to support EPGs. This package provides EPGs with two related sets of information: the program schedule on each channel and the rating information associated either with the channel or a specific program event. The ProgramSchedule object can be used to retrieve the program that is currently playing, the immediately following one, and then any other available program in the future for a specified time period. Each ProgramEvent object can be queried for its name, start time and end time, description, rating, and other related information.

Rating related information is organized into rating regions (such as countries, groups of countries or arbitrary geographical regions) where each region may have a multiple rating dimensions, such as the MPAA rating, FCC TV rating, DVB age-based rating, and broadcaster-specific rating. Each dimension contains multiple levels; each ProgramEvent object is labeled with one of these levels for all supported rating regions.

3.2.4 Transport Package

The transport package includes information about the physical media, such as MPEG-2 transport, that delivers the content the SI describes. The SI manager provides access to Transport objects, which in the MPEG case is extended by the TransportStreamCollection to represent an MPEG-2 multiplex. The generic Transport interface can be extended to support other types of transport delivery mechanisms, such as Internet Protocol (IP).

Service Selection

Once a service has been discovered, the service selection API allows an application to control the presentation of services in a simple, high level way. The service selection API combines into a single method call control of tuning, service information, media playback, broadcast file transport, and the application manager. In particular, it largely conceals the nature of the components making up the service.

For instance, an application can present a service without needing to know about its audio components, video components or subtitle components. Furthermore, an application does not need to know about whether the service has an associated application or anything about how to launch such an application.

4.1Service Selection Definitions

- **service context** - an environment in which services are presented. A receiver supports one or more service contexts. Within a service context, one service may be selected for presentation at a time.
- **service content handler** - an entity in the receiver responsible for the presentation of some portion of a service. A single service content handler may present several service components that share the same time clock. The lifecycle of the service content handler is bound by the lifecycle of the presentation of the service. An individual service content handler may also have its own lifecycle within the lifecycle of the service. For example, an application within a service may replace itself with another application in the same service.

4.2 Service Selection API Overview

The purpose of the service selection API is to provide applications with a mechanism to select services for presentation. The class that represents the environment in which these services are presented is the ServiceContext class. Receivers may limit the number of objects of this class that they support, even to one instance. ServiceContext is a high level representation of a tuner, its associated decoding hardware, and state. ServiceContext allows an application to control the presentation of the components associated with a particular service. The `select()` method on a ServiceContext object causes the service context to attempt to present a service. This selection is asynchronous and completion is notified via an event-listener mechanism. Failure to select a service is reported via an exception, if it can be determined at the time `select()` is called, or via an event, if it is determined at some time later. Once a service context is presenting a service, various information can be obtained about that service, including locators for the components of the service.

When a ServiceContext object is presenting a service, the `getServiceContentHandlers()` method returns references to the "engines" or "players" that are presenting the various service components. For real-time media such as audio, video, and subtitles, JMF Players are returned as objects of type ServiceMediaHandler. For several audio, video, and subtitle components sharing the same clock, a single JMF Player is returned.

4.3 Service Context State Model

A ServiceContext can exist in one of four states - Presenting, Not Presenting, Presentation Pending, and Destroyed. The initial state is Not Presenting. From any state (except the Destroyed state), the `select()` method can be called. Assuming no exception is thrown, the ServiceContext enters the Presentation Pending state. No event is generated on this state transition.

If a call to `select()` method completes successfully, either a NormalContentEvent or an AlternativeContentEvent is generated and the ServiceContext moves into the Presenting state. If the service selection fails, a SelectionFailedEvent is generated. If the state before the select call was Not Presenting, the ServiceContext returns to that state and a PresentationTerminatedEvent is generated. If the state before the select call was Presenting, the ServiceContext tries to return to a previous state which can result in a NormalContentEvent or AlternativeContentEvent if possible. If that is not possible, the ServiceContext returns a PresentationTerminatedEvent.

The Not Presenting state is entered due to service presentation being stopped, which is reported by the PresentationTerminatedEvent. The stopping of service presentation can be initiated by an application calling the `stop()` method or because some change in the environment makes continued presentation impossible.

The Destroyed state is entered by calling the `destroy()` method. Once this state is entered, the ServiceContext can no longer be used for any purpose. A destroyed ServiceContext is eligible for garbage collection once all references to it by any application have been removed.

FIGURE 5 shows the state machine diagram for ServiceContext objects.

FIGURE 5 ServiceContext States

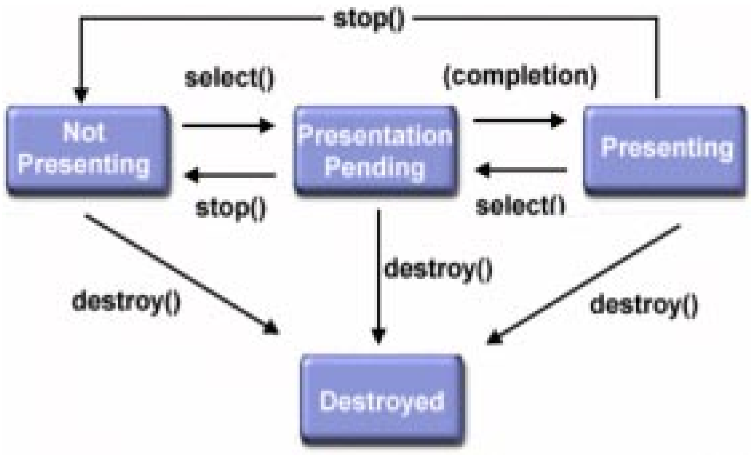


TABLE 1 describes each valid ServiceContext state.

TABLE 1 **Descriptions of the ServiceContext States**

State Name	Description
Not Presenting	The initial state of the ServiceContext. In this state, no service is presented to the viewer. The ServiceContext also enters this state if its stop method is called or if a previously presented service can no longer be presented.
Presentation Pending	The ServiceContext enters the Presentation Pending state after the select method is called and no exceptions are thrown. If a service was being presented in the previous state, the service continues to be presented in this state. If the selection operation does not complete successfully, the ServiceContext leaves the Presentation Pending state and attempts to return to its previous state.
Presenting	The ServiceContext enters the Presenting state if the service selection operation completes successfully. In this state, either normal content or alternative content is presented to the viewer.
Destroyed	The ServiceContext enters the Destroyed state when the destroy method is called. In this context, no service is presented to the viewer. Once this state is entered, the ServiceContext can no longer be used.

JMF and the Broadcast Pipeline

The Java TV API uses the Java Media Framework (JMF) 1.0 APIs for managing the broadcast media pipeline. The JMF APIs provide a foundation for the Java TV API by defining a set of APIs and a framework for displaying time-based media that are independent of transport mechanism, transport protocol, and media content type.

JMF defines `javax.media.Player`, which extends MediaHandler, for time-based media data. A Player object encapsulates the state machine required to acquire resources and manage the rendering of time-based media streams. A Player object also provides various controls for the rendering facilities (e.g., volume and video picture controls). Finally, in the case of a Player for an audio/video stream, a GUI component object that contains the video portion of the stream can be obtained from the Player. This allows easy integration and placement of video with the rest of the presentation.

For a detailed description of JMF, see the JMF 1.0 specification.

5.1 JMF Controls

A JMF Control is an object that is obtained from a Player at runtime. The object implements the `javax.media.Control` interface, and will also implement at least one interface that provides control over some aspect of the media the Player is managing. For example, many Players provide an object which supports the `javax.media.GainControl` interface to control a Player's audio gain.

In addition to the controls defined in the JMF 1.0 specification, the Java TV API includes the following controls defined in the `javax.tv.media` package.

TABLE 2 **Controls included in the Java TV API**

Interface	Function
javax.tv.media.MediaSelectControl	Media selection
javax.tv.media.AWTVideoSizeControl	Video size and position

The set of controls defined as part of the Java TV API are controls that a Java TV implementation may support, though not all Players will support all of these controls all of the time. An application can check if a Player instance supports a particular control using the Player's `getControl(String forName)` method. If the output is null, the control specified by `forName` is not supported. To obtain all the controls a Player instance supports, use the `getControls()` method. This method returns an array of Control objects that might include manufacturer specific controls, if the implementation supports them.

Additionally, the DAVIC 1.4 (Digital Audio-Video Council) specification includes a number of useful JMF Control objects that may be used with implementations of the Java TV API. These controls are listed in TABLE 3.

TABLE 3 **DAVIC Controls**

DAVIC Control	Function
org.davic.media.MediaTimeEventControl	Time based events
org.davic.media.LanguageControl	Base class for language selection
org.davic.media.AudioLanguageControl	Audio language selection
org.davic.media.SubtitlingLanguageControl	Subtitle language selection
org.davic.media.FreezeControl	Freeze frame
org.davic.media.MediaTimePositionControl	Position

5.2 JMF Synchronization

JMF allows the specification of synchronization relationships between media and the clock that serves as the synchronization master for presenting the media. The details of the synchronization primitives are described in the JMF documentation. A JMF Player inherits from the Clock class, which provides a method to obtain the current media-time. Clock also provides a method to obtain an object called a time-base. The time-base represents the synchronization master for Clock, which has methods for specifying the synchronization point between media-time and the time-base, as well as other parameters for controlling the relationship between the media-time and the time-base.

A new mechanism defined by DAVIC is supported in the Java TV API to provide for the delivery of an event at a particular media-time. A Player that can support delivery of such events provides the MediaTimeEventControl interface, which provides for the registration of MediaTimeEventListeners. The MediaTimeEvent is delivered to MediaTimeEventListeners when the appropriate media-time has occurred.

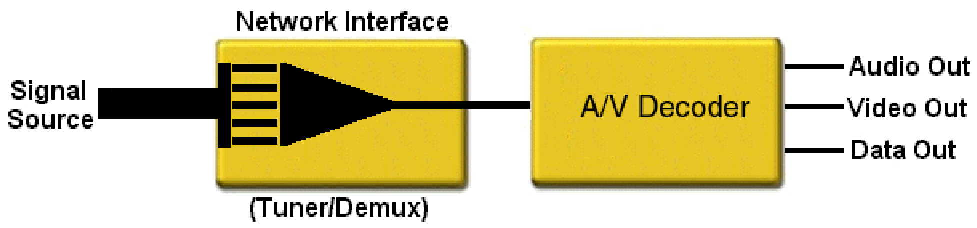
5.3 Player Architecture and the Broadcast Pipeline

JMF controls the playback of media data with an object of class `javax.media.Player`. There are two distinct components that are created for this: protocol handlers and media handlers. A Player is a type of media handler. A protocol handler is a source of data; a media handler is a consumer of data. A protocol abstracts and, therefore, is completely dependent upon, the data delivery mechanism that is used. For example, separate protocol handlers are required for HTTP delivered over an IP connection and MPEG-2 Transport delivered from a cable tuner. JMF defines the abstract class `javax.media.protocol.DataSource` as the base class for all protocol handlers. JMF defines the interface `javax.media.MediaHandler` for all content handlers; `javax.media.Player` extends MediaHandler.

Most implementations of JMF assume that a complete decoding pipeline should be constructed each time a new media stream is to be rendered. In desktop environments, this is a natural notion. Usually, separate network connections are required for each stream of media. Each connection has completely separate pipeline requirements, and requires a potentially complex negotiation with the source of the connection (e.g., a server).

In a broadcast environment, this is not the case. The interface to the broadcast network can be modeled as a multiplex of multiplexes. Such a model combines the actual tuner (the first multiplex) and demux (the second multiplex). Thus, the control interface to a broadcast network consists of tuning (primary multiplex selection) and stream selection (secondary multiplex selection) (see FIGURE 6). For traditional video broadcast, the resulting pipeline is always the same: the secondary multiplex is connected to an audio/video decoder. Thus, channel changing requires no acquisition of new pipeline resources (tuner, demux, codec, screen), merely a command to the network interface with tuning and stream selection parameters. This presumes that the same tuner and rendering sub-system is used to view the different channels.

FIGURE 6 **Broadcast Network Interface**



JMF can model this pipeline with no modification to the existing interfaces. The Player object still represents the whole pipeline and the DataSource still represents the connection to the network. To accommodate the broadcast selection mechanisms described above, a simple addition is required. Rather than require that the pipeline be reconstructed whenever one of the two multiplexes is switched (which is what happens if a new player is constructed for each tuning request), a selection interface that can affect the two demuxes is provided. This mechanism is found in `javax.tv.media.MediaSelectControl`. It is a JMF Control with APIs for asynchronous selection and de-selection of content.

Broadcast Data APIs

The Java TV APIs for broadcast data permit access to data transmitted in the television broadcast signal. These APIs support access to data carried in three formats:

- **Broadcast file systems**
  The Java TV API provides access to broadcast file and directory data through use of the file access mechanisms defined in the package `java.io`.

- **IP datagrams**
  The Java TV API provides access to unicast and multicast IP datagrams transmitted in the broadcast stream through the use of the conventional datagram reception mechanisms of the `java.net` package.

- **Streaming data**
  The Java TV API provides access to generic streaming data extracted from the broadcast using the JMF package `javax.media.protocol`.

6.1 Broadcast Data API Definitions

- **DSM-CC** - MPEG-2 Digital Storage Media Command and Control, as defined in ISO/IEC 13818-6 (see the reference to DSM-CC in Appendix I).
- **object carousel** - A mechanism for cyclic transmission of DSM-CC User-to-User (U-U) Objects over data carousel. Object carousels convey hierarchical file structures using DSM-CC U-U File and Directory objects.
- **data carousel** - A mechanism for cyclic transmission of data modules, as defined by the DSM-CC User-to-Network Download protocol.

- **asynchronous data** - data that includes no timing requirements. In an MPEG-2 transport stream, asynchronous data contains no program clock reference (PCR) or presentation time stamp (PTS) values.

6.2 Broadcast File Systems

The Java TV API provides access to broadcast file and directory data through use of the file access mechanisms defined in the package `java.io`. Such data is typically transmitted in a “carousel” wherein the contents of a remote file system are cyclically transmitted to permit reconstruction on the receiver. The Java TV API models broadcast carousels as conventional disk file systems with high access latencies. Most interactions with the specific carousel protocols are handled by the Java TV API implementation rather than the application.

The Java TV API is sufficiently high level for use with any broadcast file system protocol. However, its use with two prevalent protocols, the DSM-CC object carousel protocol and the DSM-CC data carousel protocol, are described in detail below.

6.2.1 DSM-CC Object Carousels

The DSM-CC object carousel protocol is a commonly used form of broadcast file system. It specifies three object types for structuring carousel data:

- **DSM::ServiceGateway** - provides access to the top-level directory of an object carousel.
- **DSM::Directory** - represents a conventional directory structure; may refer to files or other directories.
- **DSM::File** - represents generic file data.

The class `java.io.File` represents all of these object types. The Java TV API class `javax.tv.carousel.CarouselFile` subclasses `java.io.File` to handle object carousel access, adding the ability to:

- Refer to a carousel object using `javax.tv.locator.Locator`
- Notify applications of updates to individual carousel objects

Applications use the conventional file input classes of the `java.io` package (i.e., FileInputStream, FileReader, and RandomAccessFile) to read from a CarouselFile object.

6.2.1.1 Object Carousel Example Usage

1. Create CarouselFile of top-level directory.
```
CarouselFile serviceGateway = new CarouselFile(locator);
```

2. List top-level objects.
```
String files[] = serviceGateway.list();
```

3. Create a file object.
```
CarouselFile myFile = new CarouselFile(serviceGateway, files[0]);
```

4. Create a file input object.
```
FileInputStream fis = new FileInputStream(myFile);
```

5. Read from file.
```
byte data = fis.read();
```

6. Close file.
```
fis.close();
```

When a CarouselFile of a ServiceGateway is instantiated, the receiver "mounts" the associated carousel in the local file system, attaching the carousel's namespace as a subtree of the local file system hierarchy. The location of the mount point in the local file system is dynamically determined or specified by television standards bodies. After the carousel has been mounted, the application can query the location of the carousel in the local file system hierarchy using the method `CarouselFile.getCanonicalPath()`.

After a carousel has been mounted in the local file system, applications can access objects of type DSM::Directory and DSM::File in the carousel using CarouselFile objects or normal `java.io.File` objects. The `CarouselFile` class has special asynchronous loading features not found in `java.io.File` (see Object Carousel Management).

The read methods on the file input classes `FileInputStream`, `FileReader`, and `RandomAccessFile` throw instances of `IOException` if the requested data cannot be loaded from the carousel. If the carousel is no longer accessible, the receiver may permit applications to continue to read from previously loaded data.

6.2.1.2 Object Carousel Management

Although applications can treat object carousels much like any other file system, the receiver must interact with carousels explicitly. This interaction should be consistent from one receiver implementation to another to provide consistent behavior to applications. The following actions are recommended when implementations of the Java TV API access DSM-CC object carousels.

Upon instantiating a CarouselFile of a DSM::ServiceGateway object, the receiver:

1. Attaches the service domain of the referenced carousel,
2. Mounts the carousel file hierarchy in the local file system, and
3. Asynchronously loads the contents of the DSM::ServiceGateway object.

Upon instantiating a CarouselFile of a DSM::Directory object, the receiver asynchronously loads the contents of the DSM::Directory object. A call to the directory method `CarouselFile.listDirectoryContents()` blocks until the contents of the DSM::ServiceGateway or DSM::Directory referenced by the CarouselFile are loaded.

Likewise, instantiating a CarouselFile referencing a DSM::File object asynchronously loads the contents of the DSM::File object. A read operation on a `java.io.FileInputStream`, `java.io.FileReader`, or `java.io.RandomAccessFile` object opened on a CarouselFile object representing a DSM::File object blocks until the contents of the corresponding DSM::File object are loaded.

A close operation on every instance of `java.io.FileInputStream`, `java.io.FileReader`, and `java.io.RandomAccessFile` corresponding to a single DSM::File object unloads the contents of the DSM::File object.

Finalization of all instances of CarouselFile referring to a single DSM::Directory object unloads the contents of the DSM::Directory object. After all instances of CarouselFile referring to objects in the carousel have been finalized and all instances of `java.io.FileInputStream`, `java.io.FileReader`, and `java.io.RandomAccessFile` referring to DSM::File objects in the carousel have been closed, the receiver:

1. Unloads the DSM::ServiceGateway object,
2. Unmounts the carousel from the local file system, and
3. Detaches the service domain of the carousel.

6.2.2 DSM-CC Data Carousels

The DSM-CC data carousel protocol supports transmission of a single-directory file system to the receiver. The data carousel protocol includes a DownLoadInfoIndication message announcing the data modules present in a particular carousel, and messages to transmit the contents of each carousel module. If the television receiver is compliant with broadcast standards that permit string-based naming of data carousel modules, instances of CarouselFile can be used to access and read data carousel modules in a manner similar to that for DSM::File objects in an object carousel. Specifically,

- A CarouselFile instantiated as the top-level "directory" of a data carousel provides the contents of the DownLoadInfoIndication message for the carousel.
- Instances of CarouselFile may be created to refer to individual data carousel modules. Instantiating a CarouselFile object asynchronously loads the module to which it refers.
- A read operation on a `java.io.FileInputStream`, `java.io.FileReader`, or `java.io.RandomAccessFile` instance opened on a CarouselFile object representing a carousel module blocks until the corresponding module is loaded. The read operation accesses only the contents of the blockDataByte field of the DownloadDataBlock messages comprising the module.
- A close operation on every instance of `java.io.FileInputStream`, `java.io.FileReader`, and `java.io.RandomAccessFile` corresponding to a single data carousel module unloads the module.

6.2.3 Reducing the Effects of Carousel Latency

Access to data in DSM-CC object carousels or data carousels can be subject to a much higher degree of latency than is found in a typical disk-based file system. In the absence of measures to deal with this latency, applications that access multiple carousel-based files might experience considerable delays as all the needed data is loaded. To reduce these delays, applications based on the Java TV API can use the following latency-management techniques:

- Applications can create a new thread per carousel file to be read, and then block in parallel on read operations. This minimizes the average time required for accessing each file, but causes additional thread overhead.
- Applications can poll non-blocking status methods, such as `FileInputStream.available()` or `FileReader.ready()`, to determine the availability of data.
- Applications can time out on read operations by issuing `Thread.interrupt()` calls to blocked threads.
- Instantiating a file input class on a CarouselFile asynchronously loads the corresponding carousel data. Therefore, applications can first create instances of file input classes for each required carousel file and then block in series on read operations. This minimizes the total time required to access all the required files, but typically causes the average file access time to be greater than in the case of multiple parallel reads.

6.3 IP Datagrams

The Java TV API provides access to IP datagrams transmitted in the broadcast stream through use of the normal datagram reception mechanisms of the `java.net` package. Applications receive unicast IP datagrams using the `java.net.DatagramSocket` class. Applications receive multicast IP datagrams using the `java.net.MulticastSocket` class.

To enable reception of multicast IP datagrams, the Java TV API assigns a locally-unique IP address to service components carrying encapsulated IP datagrams. These addresses are generated dynamically from the set of IP addresses reserved for use in private networks (see RFC 1918 referenced in Appendix I for more information). Television applications determine the local IP address assigned to a given service component using the class `javax.tv.net.InterfaceMap`. Applications then use this IP address to indicate the network interface from which instances of `java.net.MulticastSocket` or `java.net.DatagramSocket` receive multicast datagrams.

6.4 Streaming Data

The Java TV API provides access to generic streaming data in the television broadcast using the JMF package `javax.media.protocol`. Asynchronous streaming data can be obtained using the interface `javax.tv.media.protocol.PushSourceStream2`.

A Java TV API application typically refers to an individual data service component using a `javax.tv.locator.Locator` object. Using the method `Locator.toExternalForm()`, an application converts the Locator object into a string from which a `javax.media.MediaLocator` object is constructed. The resulting MediaLocator object is then used to obtain a `javax.media.DataSource` object from `javax.media.Manager`. Then, the DataSource object is used to obtain one or more PushSourceStream2 objects.

The interface PushSourceStream2 extends the JMF version 1.0 interface `javax.media.protocol.PushSourceStream` with a new read mechanism. The method `PushSourceStream2.readStream()` provides access to the payload of the data and throws exceptions to indicate data loss. The Java TV API makes no guarantees concerning buffering or availability of the data obtained through PushSourceStream2.

Application Lifecycle

The Java TV API defines an application model called the Xlet application lifecycle. Java applications that use this lifecycle model are called Xlets. The Xlet application lifecycle is compatible with the existing application environment and virtual machine technology.

The Xlet application lifecycle model defines the dialog (protocol) between an Xlet and its environment through the following:

- A simple, well-defined state machine
- A concise definition of the application's states
- An API to signal changes between the states

7.1 Xlet Application Lifecycle Definitions

The following definitions are used in the Xlet application lifecycle model:

- **application manager** - A part of a digital television receiver's software operating environment that manages Java applications. The application manager controls the lifecycle of an Xlet by signalling its state changes. An application manager is required on a receiver, but its precise behavior is implementation specific.
- **Xlet** - A Java application (usually downloaded) that runs on the digital television receiver.
- **Xlet states** - The states changes of an Xlet are handled by the Xlet itself, i.e., only the Xlet knows when the state has been successfully changed. The four Xlet states are Loaded, Active, Paused, and Destroyed. Xlets communicate with the application manager about state changes via callbacks. The Xlet signals the success or failure of such changes with the return value of the callbacks.

- **Xlet context** - the object that an Xlet uses to access other facilities in the system. Each Xlet has one XletContext object and it can be tailored to a specific environment.

7.2 Application Manager Requirements

The Xlet application lifecycle addresses the amount of control that an application manager can exert over an Xlet. The application manager's control over Xlets does not include giving an Xlet access to other resources on the receiver, such as graphics or shared resource allocation/management. Note that the application manager may or may not be written entirely in the Java language.

Although a detailed specification of an application manager is outside the scope of the Java TV API, the Xlet application lifecycle model requires that the resident application manager adhere to the following principles:

- **An Xlet can be destroyed at any time.**
  An application manager is the entity on a digital television receiver that has ultimate control over the Xlets it manages. Therefore, the application manager must be able to destroy an Xlet at any time.

- **The current state an Xlet will always be known.**
  An application manager is responsible for signaling Xlets regarding their current state. Xlets, however, can also change their own states, but they must signal those changes back to the application manager.

- **An application manager can change the state of an Xlet.**
  The primary purpose of an application manager is to direct the state changes of an Xlet.

- **An application manager will know if an Xlet has changed its state.**
  One of the features of the Xlet application lifecycle API is that the Xlet can change its own state. Therefore, the application manager must be notified of this state change so it can track the state of the Xlet.

7.3 Xlet States

The lifecycle states for Xlets are:

TABLE 4 Xlet States

State Name	Description
Loaded	The Xlet has been loaded and has not been initialized. This state is entered after the Xlet has been created using new. The no-argument constructor for the Xlet is called and returns without throwing an exception. The Xlet typically does little or no initialization in this step. If an exception occurs, the Xlet immediately enters the Destroyed state and is discarded. Note: This state is entered only once per instance of an Xlet.
Paused	The Xlet is initialized and quiescent. It should not be holding or using any shared resources. This state is entered: From the Loaded state after the Xlet.initXlet() method returns successfully, or From the Active state after the Xlet.pauseXlet() method returns successfully, or From the Active state before the XletContext.notifyPaused() method returns successfully to the Xlet.
Active	The Xlet is functioning normally and providing service. This state is entered from the Paused state after the Xlet.startXlet() method returns successfully.
Destroyed	The Xlet has released all of its resources and terminated. This state is entered: When the destroyXlet() method for the Xlet returns successfully. The destroyXlet() method shall release all resources held and perform any necessary clean up so it may be garbage collected; or When the XletContext.notifyDestroyed() method returns successfully to the Xlet. The Xlet must perform the equivalent of the Xlet.destroyXlet() method before calling XletContext.notifyDestroyed. Note: This state is only entered once per instance of an Xlet.

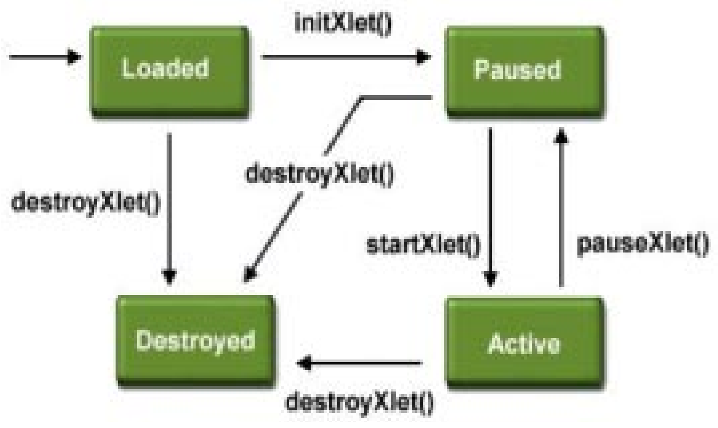
7.3.1 Xlet State Machine

The Xlet state machine is designed to ensure that the behavior of an Xlet is as close as possible to the behavior television viewers expect, specifically:

- The perceived startup latency of an Xlet should be very short.
- It should be possible to temporarily stop an Xlet from providing it service.
- It should be possible to destroy an Xlet at any time.

FIGURE 7 shows the application state machine diagram for Xlets.

FIGURE 7 **Xlet State Machine Diagram**



7.3.2 Xlet Lifecycle Model

Only the Xlet can determine if it is able to provide the service for which it was designed. Therefore, an application manager cannot force an Xlet to provide its service; it can only indicate that the Xlet is permitted to do so. A typical sequence of Xlet execution is:

TABLE 5 **Xlet Execution**

Application Manager	**Xlet**
The application manager creates a new instance of an Xlet.	The default (no argument) constructor for the Xlet is called; it is in the Loaded state.
The application manager creates the necessary context object for the Xlet to run and initializes the Xlet.	The Xlet uses the context object to initialize itself. It is now in the Paused state.
The application manager has decided that it is an appropriate time for the Xlet to perform its service, so it signals it to enter the Active state.	The Xlet acquires any resources it needs and begins to perform its service.
The application manager no longer needs the Xlet to perform its service, so it signals the Xlet to stop performing its service.	The Xlet stops performing its service and might choose to release some resources it currently holds.

Application Manager	Xlet
The application manager has determined that the Xlet is no longer needed, or perhaps needs to make room for a higher priority application, so it signals the Xlet that it is to be destroyed.	If it has been designed to do so, the Xlet saves state or user preferences and performs clean up.

7.4 Xlet Package

The Xlet package provides developers with APIs that provide application lifecycle signaling in a digital television receiver environment. The Xlet API consists of two interfaces, Xlet and XletContext, which express the communication between an Xlet and its environment. (See Javadocs for more details.)

The Xlet APIs use a callback approach to signal state changes. The state of an Xlet can change by either the application manager calling one of the methods on Xlet or by the Xlet notifying the application manager of an internal state transition via the XletContext object. The semantics of exactly when that state change occurs are important:

- **Calls to Xlet**
  Calls to this interface indicate a successful state change only when the call successfully returns.

- **Calls to XletContext**
  Calls to this interface indicate a state change on entry.

The Xlet APIs adhere to the following principles:

- **The Xlet API signals an Xlet when a state change is required.**
  The primary purpose of the Xlet API is to direct the state changes of an Xlet.

- **A context will be provided to the Xlet when it is initialized.**
  An XletContext is an object that is used to represent the Xlet. An XletContext is passed to the Xlet at initialization to permit configuration based on the environment.

- **An Xlet can signal when it has changed state.**

An individual Xlet is the only entity that can define whether or not it is able to perform properly. Therefore, an Xlet may discover that it can no longer operate as desired and may choose to change its state.

- **An Xlet can signal when it is done.**
  When the Xlet has completed its task, it should signal this to the application manager.

7.4.1 Xlet Interface

The Xlet interface provides an application manager with four methods to signal application lifecycle state changes to an Xlet:

- `public void initXlet(XletContext ctx)`
  Initializes the Xlet. This method is a signal for Xlet to initialize itself, so that it is prepared to provide its service quickly. An XletContext object is passed in with this method. This object can be used by the Xlet to access properties associated with its environment, as well as to signal the application manager that it has changed state. If for some reason the Xlet cannot successfully initialize, it can signal this to the application manager by throwing an XletStateChangeException. Otherwise, the Xlet returns the Paused state.

- `public void startXlet()`
  The Xlet moves to the Active state when this method completes. The Xlet should now be providing its service. Xlets typically attempt to acquire resources at this time. If the Xlet cannot enter the Active state, it can throw an XletStateChangeException, which will notify the application manager that the state change failed.

- `public void pauseXlet()`
  The pauseXlet callback signals the Xlet to stop providing service. When the callback returns, the Xlet is in the Paused state. The Xlet may choose to release some resources at this time. If the Xlet cannot enter the Paused state, it can throw an XletStateChange-Exception, which will notify the application manager that the state change failed.

- `public void destroyXlet()`
  This method is a signal to the Xlet that it is no longer needed and that it will soon be purged from the system. Xlets should perform any operations required before termination, such as releasing resources, saving preferences, and saving state.

7.4.1.1 Xlets and Finalization

The Java language provides a mechanism called finalization that allows objects to perform some clean up just before they are garbage collected. The finalizer on an object will not be called until all references to the object have been discarded and the object is ready to be garbage collected. The Java language specification states that programmers should never depend on a finalizer being called. Note that in the Xlet interface the `destroyXlet()` method is called on an Xlet shortly before it is to be destroyed. The intent of the two methods are similar, but programmers can assume that the `destroyXlet()` method will be called.

- **What Xlets should do in their finalizers.**
  As mentioned above, objects should not depend on their finalizers being called. If an Xlet must typically do clean up at the end of normal execution, the Xlet should use its `destroyXlet()` method instead. In most cases the finalizer should be empty.

- **Reducing the risk of misbehaving Xlets.**
  In general, the underlying Java application environment should be designed to reduce the risk of Xlets that misbehave in their finalizers. One possible technique is to set the maximum priority of the finalizer's thread group very low. Constructors on objects thought to cause potential risks (such as Thread and ThreadGroup) can also throw security exceptions. The implementor of the application environment must provide a SecurityManager that implements this policy. An application environment will likely provide at least two SecurityManager objects, one for the normal operation of an Xlet and one that implements the security policy of the ThreadGroup objects that execute the finalizer threads.

7.4.1.2 Xlets and Threads

Xlets should declare their lifecycle methods (the methods implementing the Xlet interface) as `synchronized`. This is not done in the Xlet interface method signatures because interface methods cannot be declared synchronized in the Java language. By declaring these methods synchronized, a lock will be acquired on the Xlet object interface when one of the lifecycle methods is called. This will have the effect of blocking other calls to the other method's lifecycle methods. Xlets can also benefit from acquiring a lock on themselves before calling the lifecycle methods on their XletContext. A well-designed application manager should create individual threads to call the lifecycle methods on Xlets.

7.4.2 XletContext Interface

An XletContext is an object passed to an Xlet when it is initialized. XletContext provides an Xlet with a mechanism to obtain properties, as well as a way to signal internal state changes to the application manager. There is a one-to-one correspondence between XletContext objects and Xlets. The XletContext interface defines the following methods:

- `public void notifyDestroyed()`
  This method signals the application manager that the Xlet has entered the Destroyed state. This method allows an Xlet to signal the application manager that the Xlet has completed execution and is ready to be destroyed.

- `public void notifyPaused()`
  This method signals the application manager that the Xlet has entered the Paused state. This state is entered when an Xlet can no longer provide its service.

- `public java.lang.Object getXletProperty(java.lang.String key)`
  This method allows an Xlet to retrieve named properties from the XletContext.

- `public void resumeRequest()`
  This method signals the application manager that the Xlet is interested in entering the Active state.

7.5 Xlet Lifecycle Example

A simple example of Xlet lifecycle is a stock ticker that uses a back channel to retrieve stock quotes, which it displays on the viewer's television.

1. The application manager obtains the code for the Xlet.
2. The application manager creates an instance of the XletContext object and initializes it for the new Xlet.
3. The application manager initializes the Xlet by calling its `initXlet()` method and passing it the XletContext object.
4. The Xlet uses the XletContext object to initialize itself and enters the Paused state.
5. The user presses a button on the television's remote control that signals the application manager to start the Xlet.

6. The application manager calls the `startXlet()` method for the Xlet. The application manager assumes that the Xlet is performing its service.
7. Upon receiving this signal, the Xlet creates a new thread that opens the back channel to retrieve the stock quotes. The Xlet is now in the Active state.
8. The Xlet begins to show the stock quotes.
9. Due to circumstances beyond the control of the Xlet, it is no longer able to retrieve updated stock quotes.
10. The Xlet decides to continue displaying the most recent quotes it has. Note that the application is still in the Active state.
11. After a time, the Xlet is still unable to open the back channel. It decides that the quotes it is displaying are too old to present and that it can no longer perform its service. It chooses to take itself out of the Active state. It calls the `notifyPaused()` method on the XletContext object to signal this change to the application manager.
12. Finally, the Xlet decides it no longer has any chance of performing its service, so it decides it should be terminated. The Xlet does some final clean up and calls the `notifyDestroyed()` method on the XletContext object to signal the application manager that it has entered the Destroyed state.
13. The application manager prepares the Xlet for garbage collection.

Appendix I: Related Documents

The PersonalJava application environment programming interfaces are specified by the *PersonalJava API Specification, Version 1.2* at http://java.sun.com/products/personaljava. This specification defines the relationship between the PersonalJava API and the JDK^{TM} API.

The *Java Platform 1.1 Core API Specification* describes the Java platform and is available at http://java.sun.com/products/jdk/1.1/docs/api/packages.html.

The Java programming language is described in *The Java Programming Language, Second Edition* (ISBN: 0-201-31006-6) by Ken Arnold and James Gosling at http://java.sun.com/docs/books/javaprog/secondedition. This book covers the constructs of the language and core packages in detail.

The Java programming language is specified in *The Java Language Specification* (ISBN 0-201-63451-1) by James Gosling, Bill Joy, and Guy Steele at http://java.sun.com/docs/books/jls.

The Java virtual machine is specified in *The Java Virtual Machine Specification* (ISBN 0-201-63452-X) by Tim Lindholm and Frank Yellin at http://java.sun.com/docs/books/vmspec. This book contains comprehensive coverage of the Java virtual machine class file format and instruction set. In addition, the book contains directions for compiling the JVM with numerous practical examples to clarify how it operates in practice. The book also demonstrates the VM's powerful verification techniques.

The JDK 1.1 class libraries are described in two volumes. The first volume, *The Java Class Libraries: Second Edition, Volume 1* (ISBN 0-201-31002-1) by Patrick Chan, Rosanna Lee, and Douglas Kramer, is at http://java.sun.com/docs/books/chanlee/second_edition/vol1. The second volume*, The Java Class Libraries: Second Edition, Volume 2* (ISBN 0-

201-31003-1) by Patrick Chan and Rosanna Lee, is at http://java.sun.com/docs/books/chanlee/second_edition.

The Secure Sockets Layer (SSL) is documented in *SSL Java Standard Extension to JDK 1.1* at http://java.sun.com/security/.

The Java Media Framework is specified in the *Java Media Framework 1.0 Specification* at http://java.sun.com/products/java-media/jmf/forDevelopers/playerapi/packages.html. This specification documents the APIs. A developer's guide describing the Java Media Framework is available in Java Media Players at http://java.sun.com/products/java-media/jmf/forDevelopers/playerguide/index.html.

The Digital Audio-Visual Industry Consortium (DAVIC) has defined Java language APIs for digital television. These can be found at ftp://ftp.davic.org/Davic/Pub/Spec1_4/14p09ml.zip in part 9.

For broadcast data, the MPEG Systems Layer is described in ISO/IEC 13818-1: *Information technology - Generic coding of moving pictures and associated audio information - Part 1: Systems*.

DSM-CC is defined in ISO/IEC 13818-6: *Information technology - Generic coding of moving pictures and associated audio information - Part 6: Extension for Digital Storage Media Command and Control (DSM-CC)*.

IP multicasting is described further in RFC 1112 *Host Extensions for IP Multicasting*, August 1989, by S. Deering; and RFC 2365 *Administratively Scoped IP Multicast*, July 1998, by D. Meyer.

IP addresses reserved for use in private networks are described in RFC 1918 *Address Allocation for Private Networks*, February 1996, by Y. Rekhter, et al.

For broadcast data, the Internet RFCs can be found at http://www.ietf.org/rfc.html.

Index