TECHNICAL REPORT

ISO/IEC TR 13066-6

First edition
2014-07-15

Information technology — Interoperability with assistive technology (AT) —

Part 6: Java accessibility application programming interface (API)

Technologies de l'information — Interopérabilité avec les technologies d'assistance —

Partie 6: Interface de programmation d'applications (API) d'accessibilité Java

Reference number
ISO/IEC TR 13066-6:2014(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, Subcommittee SC 35, *User interfaces*.

ISO/IEC 13066 consists of the following parts, under the general title *Information technology — Interoperability with assistive technology (AT)*:

— *Part 1: Requirements and recommendations for interoperability*

— *Part 2: Windows accessibility application programming interface (API)*

— *Part 3: IAccessible2 accessibility application programming interface (API)*

— *Part 4: Linux/UNIX graphical environments accessibility application programming interface (API)*

— *Part 6: Java accessibility application programming interface (API)*

Introduction

Assistive technology (AT) is specialized information technology (IT) hardware or software that is added to or incorporated within a system that increases accessibility for an individual. In other words, it is special purpose IT that interoperates with another IT product enabling a person with a disability to use the IT product.

Interoperability involves the ability to add or replace Assistive Technology (AT) to existing components of Information Technology (IT) systems. Interoperability between AT and IT is best facilitated via the use of standardized, public interfaces for all IT components.

This part of ISO/IEC 13066 describes the Java accessibility API that can be used as a framework to support software to software IT-AT interoperability on the multiple computing platforms. It also describes the Java Access Bridge for Windows – for enabling AT on Windows to interoperate with accessible Java applications on the Microsoft Windows platform – and the Java Access Bridge for GNOME – for enabling AT on UNIX and GNU/Linux platforms running the GNOME graphical desktop to interoperate with accessible Java applications on UNIX and GNU/Linux environments.

NOTE 1 GNOME is both a common and accessible graphical desktop for Linux / UNIX graphical environments, as well as an open source project delivering a collection of software libraries and applications. It was formerly an acronym meaning “GNU Network Object Model Environment”.

NOTE 2 The code examples contained in this document are illustrative in nature. With rare exception, they do not include error checking or exception handling, and should be treated more like pseudo-code than as cookbook templates that can use directly in applications or assistive technologies.

Information technology — Interoperability with assistive technology (AT) —

Part 6:
Java accessibility application programming interface (API)

1 Scope

This part of ISO/IEC 13066 provides an overview to the structure and terminology of the Java accessibility API

It will provide:

- A description of the overall architecture and terminology of the API;
- Further introductory explanations regarding the content and use of the API beyond those found in Annex A of ISO/IEC 13066-1;
- An overview of the main properties, including of:
  - user interface elements;
  - how to get and set focus;
  - of communication mechanisms in the API;
  - a discussion of design considerations for the API (e.g. pointers to external sources of information on accessibility guidance related to using the API);
  - information on extending the API (and where this is appropriate);
  - an introduction to the programming interface of the API (including pointers to external sources of information).
  - an introduction to the Java Access Bridge for Windows and the Java Access Bridge for GNOME

It will provide this information as an introduction to the Java API to assist:

- IT system level developers who create custom controls and/or interface to them;
- AT developers involved in programming "hardware to software" and "software to software" interactions

2 Terms and Definitions

For the purposes of this document, the following terms and definitions apply.

2.1
accessible object
a part of the user interface that is accessible by and exposes the Java accessibility API

Note 1 to entry An accessible object is represented by an object of the “AccessibleContext” Java class

2.2
application programming interface
API
collection of invocation methods and associated parameters used by one piece of software to request actions from another piece of software

[SOURCE: ISO/IEC 18012-1 Information technology – Home electronic system – Guidelines for product interoperability – Introduction, definition 3.1.1]

2.3
application software
software that is specific to the solution of an application problem

[SOURCE: ISO/IEC 2381-1, definition 10.04.01]

EXAMPLE A spreadsheet program is application software.

2.4
Assistive Technology
(AT)
hardware or software that is added to or incorporated within a system that increases accessibility for an individual

EXAMPLE Braille displays, screen readers, screen magnification software and eye tracking devices are assistive technologies.

[SOURCE: ISO 9241-171, definition 3.5]

Note 1 to entry Within this document, where Assistive Technology (and its abbreviation AT) is used, it is to be considered as both singular and plural, without distinction. If it is to be used in the singular only, it will be preceded by the article "an" (i.e. an Assistive Technology). If it is to be used in the plural only, it will be preceded by the adjective "multiple" (i.e. multiple AT).

2.5
class
a term from object oriented programming, also used in the Java programming language, denoting the definition/description of an object containing code (methods) and data (fields)

EXAMPLE All objects in object oriented programming belong to a class (e.g. a specific window object is an instance of the window class).

Note 1 to entry Much of the Java accessibility API consists of these class definitions, and implementations of the Java accessibility API are instances of these classes.

Note 2 to entry In objected oriented programming – and specifically in the Java programming language – classes can be “subclassed” (e.g. a dialog box class is a subclass of the more generic window class), and portions of the Java accessibility API are implemented as subclasses (e.g. AccessibleRole, AccessibleState, and AccessibleRelation are all subclasses of the more generic AccessibleBundle class).

2.6
compatibility
the capability of a functional unit to meet the requirements of a specified interface without appreciable modification

[SOURCE: ISO/IEC 2381-1, definition 10.06.11]

2.7
information/communication technology
(ICT)
technology for gathering, storing, retrieving, processing, analysing and transmitting information

[SOURCE: ISO 9241-20, definition 3.4]

EXAMPLE A computer system is a type of ICT.2.13

2.8
interface
<general software> a shared boundary between two functional units, defined by various characteristics pertaining to the functions, physical interconnections, signal exchanges, and other characteristics, as appropriate

[SOURCE: ISO/IEC 2381-1, definition 10.01.38]

2.9
interface
<Java programming language> in object oriented programming generally – and the Java language in particular – an interface is a set of public methods (and potentially public fields) that all objects implementing the interface must include

Note 1 to entry As with Java programming language classes that can be "subclassed", interfaces can be "subclassed" as well – and the result is called a "subinterface".

Note 2 to entry Much of the Java accessibility API is implemented as Java interfaces, and some of these as subinterfaces (e.g. the `AccessibleEditableText` interface is a subinterface of the more generic `AccessibleText` interface).

2.10
interoperability
the capability to communicate, execute programs, or transfer data among various functional units in a manner that requires the user to have little or no knowledge of the unique characteristics of those units

[SOURCE: ISO/IEC 2381-1, definition 10.01.47]

2.11
inter-process communication
(IPC)
a mechanism by which different software processes communicate with each other – across process boundaries, runtime environments, and sometimes also computers and operating systems

2.12
operating system
(OS)
software that controls the execution of programs and that may provide services such as resource allocation, scheduling, input-output control, and data management

Note 1 to entry Although operating systems are predominantly software, partial hardware implementations are possible.

[SOURCE: ISO/IEC 2381-1, definition 10.04.08]

2.13
package
<Java programming language> a collection of class and interface definitions that are related to one another, and which are bundled together (into a package)

EXAMPLE The Java accessibility API is collected together into the core Java platform package `javax.accessibility.*` and the fully qualified name of each class or interface in this package begins with the package name, e.g. `javax.accessibility.AccessibleEditableText`.

2.14
runtime environment
a software environment that provides all of the resources necessary for software applications to run, yet is not itself an operating system

EXAMPLE 1 The Java runtime environment.

EXAMPLE 2 Adobe Flash player.

EXAMPLE 3 Microsoft Silverlight's runtime.

Note 1 to entry Virtual machines are type of runtime environment, which are explicitly emulating one or more specific sets of hardware.

2.15
software
all or part of the programs, procedures, rules, and associated documentation of an information processing system

Note 1 to entry Software is an intellectual creation that is independent of the medium on which it is recorded.

[SOURCE: ISO/IEC 2381-1, definition 10.01.08]

2.16
user interface
(UI)
mechanisms by which a person interacts with a computer system

Note 1 to entry The user interface provides input mechanisms, allowing users to manipulate a system. It also provides output mechanisms, allowing the system to produce the effects of the users' manipulation.

2.17
user interface element
user interface object
user interface component
entity of the user interface that is presented to the user by the software

[SOURCE: ISO 9241-171 definition 3.38]

Note 1 to entry User interface elements may or may not be interactive.

Note 2 to entry Both entities relevant to the task and entities of the user interface are regarded as user interface elements. Different user interface element types are text, graphics and controls. A user interface element may be a representation or an interaction mechanism for a task object (such as a letter, a sales order, electronic parts, or a wiring diagram) or a system object (such as a printer, hard disk, or network connection). It may be possible for the user to directly manipulate some of these user interface elements.

EXAMPLE 1 User interface elements in a graphical user interface include such things as basic objects (such as window title bars, menu items, push buttons, image maps, and editable text fields) or containers (such as windows, grouping boxes, menu bars, menus, groups of mutually-exclusive option buttons, and compound images that are made up of several smaller images).

EXAMPLE 2 User interface elements in an audio user interface include such things as menus, menu items, messages, and action prompts.

EXAMPLE 3 User interface elements in tactile interfaces include such things as tactile dots, tactile bars, surfaces, knobs, and grips.

3 General Description

3.1 General Description

The Java accessibility API was developed by Sun Microsystems, Inc. as part of the Java Foundation Classes (along with the "Swing" user interface library), and was then folded into the Java Platform release 1.2. Work began in early 1997, based on requirements gathered from industry and assistive technology stakeholders, and with review and design feedback from many of these stakeholders as it matured. The 1.0 release shipped with the Java Foundation Classes release 1.0, in 1997, and was folded into the Java SE platform in January 1998. An initial implementation of the API in the "Swing" library was also part of that release.

The Java Accessibility API was born out of necessity – the first screen access techniques for graphical systems of the Macintosh and Windows reverse-engineered and hooked into the graphics rendering pipeline to build off-screen models for screen readers (also known as "screen scraping") – techniques that would not work in the Java environment. The Java accessibility API was the first comprehensive accessibility API (a "3rd generation accessibility API"). It provided support for everything that a screen reader needed, and is the progenitor of the UNIX accessibility API (described in ISO/IEC 13066-4) and the UNO accessibility API of Oracle Open Office (which is the basis for IAccessible2, described in ISO/IEC 13066-3). It was also a model for the WAI-ARIA specification (described in ISO/IEC 13066-5).

Because the Java platform is commonly running on top of some other, underlying operating system (e.g. Microsoft Windows or Solaris or GNU/Linux or Macintosh), and users with significant disabilities are using assistive technologies designed to work with the underlying operating system, a key facet of AT-IT interoperability on the Java platform is the use of a "bridge", which exposes the Java accessibility API outside of the Java platform, and to assistive technologies running on the underlying operating system. While it is certainly possible to use AT directly within the Java platform – and such technologies have been created – it is rarely used in this fashion.

3.2 Architecture

The Java accessibility API is based on the Java object model. The API itself is contained in a Java package (`javax.accessibility.*`), that is a core part of the Java platform. User interface components that are accessible must directly implement the `javax.accessibility.Accessible` interface (and for the rest of this document we will drop the package name and simply use the class or interface names, e.g. `Accessible`). When requested by an AT (or by a bridge on behalf of the AT), the accessible user interface component must then return an object that implements the Java accessibility API (the `Accessible.getAccessibleContext()` method). This object then handles all accessibility API calls on behalf of the user interface component. This architecture means that implementation of the Java accessibility API can either be implemented directly by that object, or be "delegated" to some other object or library.

In addition to this "delegation" model, the Java accessibility API is implemented as a core `AccessibleContext` object – containing all of the information common to every user interface component – and then a set of accessibility "sub-interfaces" or "specializations" which are implemented only as appropriate for the user interface component in question. For example, components containing text would implement the `AccessibleText` optional interface (and more specifically, the `AccessibleEditableText` optional interface if that text were editable). Components which take on one of a range of values would implement the `AccessibleValue` optional interface. etc.

The diagram below shows the Java Foundation Classes user interface component `javax.swing.JSlider`, and those aspects of the Java Accessibility API implemented for it:

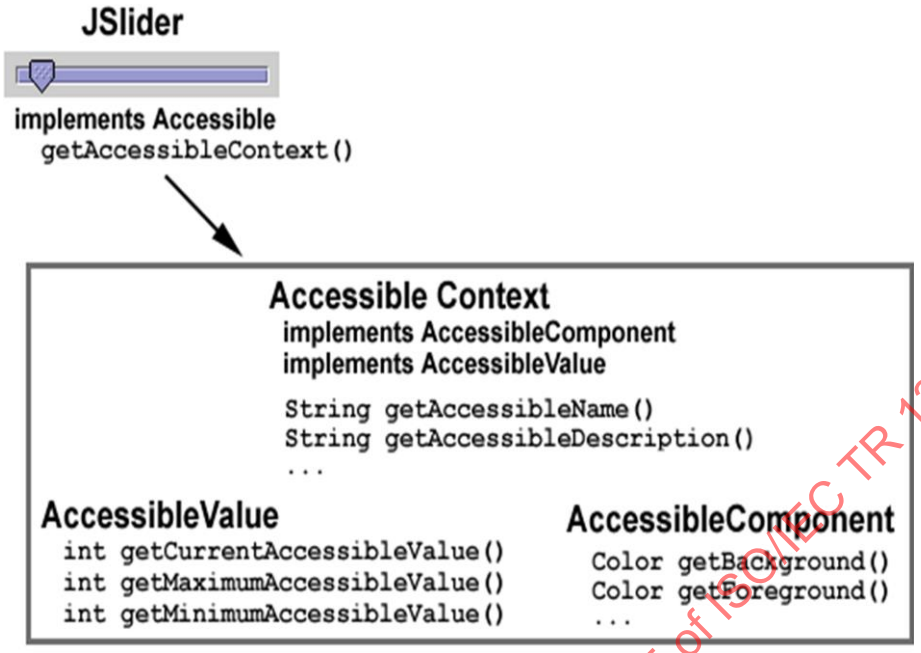

Figure 1 – Illustration of the accessibility interfaces implemented by the Swing JSlider component

4 Using the API

4.1 Overview

The Java accessibility API is contained in the `javax.accessibility.*` package. It consists of a core/tagging interface that all accessible user interface components must implement, a core accessibility class containing that portion of the Java accessibility API that all accessible user interface components must provide, an optional set of accessibility sub-interfaces, and then a set of helper classes. These are described below in the subclause

Applications can implement the Java Accessibility API in one of three ways:

a) They can use user interface components that have already implemented the Java accessibility API (such as those in the Java Foundation Classes or “Swing” library);

b) They can create one or more custom user interface components that derive from (or “subclasses”) an existing user interface component that already implements the Java accessibility API (such as a custom button that subclasses `javax.swing.JButton`, or a completely custom component that nonetheless subclasses `javax.swing.JComponent`);

c) They can create one or more custom user interface components “from scratch” and implement the necessary interfaces and methods to respond to the AT requests.

These distinct ways of implementing the API are described below.

4.2 Package javax.accessibility*

This package contains the Java Accessibility API.

The external entry point into the API is the interface Accessible, which contains the single method `getAccessibleContext()`. Every accessible user interface component must implement this interface, and must return a valid `AccessibleContext` when asked, which will implement the accessibility API on behalf of that component.

4.2.1 The AccessibleContext class

The AccessibleContext class contains the core implementation of the accessibility API. It is implemented as a class with a minimal existing implementation, with the expectation that it will be subclassed for each type of user interface component that uses it to implement the Java accessibility API. This class contains the core methods for traversing the UI hierarchy (several of which support requirements in ISO/IEC 13066-1, subclause 7.1.7(d)(1)):

— `getAccessibleParent()`

— `getAccessibleIndexInParent()`

— `getChild()`

— `getChildrenCount()`

basic information common to all user interface components (several of which support requirements in ISO/IEC 13066-1, subclause 7.1.7(a)):

— `getAccessibleName()`

— `getAccessibleDescription()`

— `getAccessibleRole()`

— `getAccessibleStateSet()`

— `getLocale()`

relationship information (which supports requirements in ISO/IEC 13066-1, subclause 7.1.7(d)(1)):

— `getAccessibleRelationSet()`

event tracking support (which support requirements in ISO/IEC 13066-1, subclause 7.1.10):

— `addPropertyChangeListener()`

— `removePropertyChangeListener()`

— `firePropertyChange()`

and several utility functions designed only to be called by implementations of the Java accessibility API:

— `setAccessibleName()`

— `setAccessibleDescription()`

— `setAccessibleParent()`

It also contains “getters” for the various optional interfaces that apply only to certain types of user interface components. These optional interfaces are described in the paragraphs below. Though it is a common practice to simply implement the appropriate optional interfaces directly on the returned `AccessibleContext` object, it is important for assistive technologies to use the getters to retrieve them, rather than using the `instanceOf()` pattern.

4.2.2 The AccessibleAction interface

This optional interface must be implemented for all user interface components that can perform one or more actions, and provides the standard mechanism for an assistive technology to determine what those actions are as well as to tell the component to perform the action programmatically.

The interface has three methods, and also defines several string constants naming common actions (to be returned in the `getAccessibleActionDescription()` call. The methods (which support requirements in ISO/IEC 13066-1, subclause 7.1.8) are:

— `doAccessibleAction()`

— `getAccessibleActionCount()`

— `getAccessibleActionDescription()`

The defined action descriptions are: `CLICK`, `DECREMENT`, `INCREMENT`, `TOGGLE_EXPAND`, and `TOGGLE_POPUP`. Additional action descriptions can be added by convention, and then folded into the platform.

4.2.3 The AccessibleComponent and AccessibleExtendedComponent interfaces

The `AccessibleComponent` optional interface roughly parallels the AWT Component interface from the Java Foundation Classes. It should be implemented for all user interface components that are drawn onto the screen. The `AccessibleExtendedComponent` interface is a subinterface of `AccessibleComponent` and adds a few things that proved to be important to assistive technologies . It should be implemented by most user interface components that are drawn on the screen (anything that has tooltip text, or a keyboard accelerator or a titled border),

Presented in logical groups, these are the methods of the `AccessibleComponent` interface:

boundary-related methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(a)):

— `getBounds()`

— `getLocation()`

— `getLocationOnScreen()`

— `getSize()`

— `setBounds()`

— `setLocation()`

— `setSize()`

visibility and visual appearance related methods (some of which support requirements in ISO/IEC 13066-1, subclause 7.1.7(c) – specifically for simple text components that don't implement the optional `AccessibleText` interface):

— `getBackground()`

— `getCursor()`

— `getFont()`

— `getFontMetrics()`

— `getForeground()`

— `isShowing()`

— `isVisible()`

— `setBackground()`

— `setCursor()`

— `setFont()`

— `setForeground()`

— `setVisible()`

focus-related methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.9):

— `addFocusListener()`

— `isFocusTraversable()`

— `removeFocusListener()`

— `requestFocus()`

parent-child location related methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(d)(3)):

— `contains()`

— `getAccessibleAt()`

interactivity related methods (one of which supports some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(a)):

— `isEnabled()`

— `setEnabled()`

These are the methods of the `AccessibleExtendedComponent` interface:

— `getAccessibleKeyBinding()`

— `getTitledBorderText()`

— `getToolTipText()`

4.2.4 The `AccessibleIcon` interface

The `AccessibleIcon` optional interface should be implemented by those user interface components that are an icon or include an icon within them (e.g. a toolbar button that is an image). These are the methods of the `AccessibleIcon` interface:

— `getAccessibleIconDescription()`

— `getAccessibleIconHeight()`

— `getAccessibleIconWidth()`

— `setAccessibleIconDescription()`

4.2.5 The AccessibleSelection interface

The `AccessibleSelection` optional interface should be implemented by those user interface components that have children user interface components inside them which can be selected (e.g. a list box or a menu). These are the methods of the `AccessibleSelection` interface (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.8):

— `addAccessibleSelection()`

— `clearAccessibleSelection()`

— `getAccessibleSelection()`

— `getAccessibleSelectionCount()`

— `isAccessibleChildSelected()`

— `removeAllAccessibleSelection()`

— `selectAllAccessibleSelection()`

4.2.6 The AccessibleStreamable interface

The `AccessibleStreamable` optional interface should be implemented by those user interface components that provide a user interface to streaming media (e.g. to HTML or a bitmap impage or MathML). This allows assistive technologies which are designed to directly parse certain types of streaming data to do so directly. These are the methods of the `AccessibleStreamable` interface:

— `getMIMETypes()`

— `getStream()`

4.2.7 The AccessibleTable and AccessibleExtendedTable interfaces

The `AccessibleTable` optional interface (and the `AccessibleExtendedTable` optional subinterface) should be implemented by those user interface components that present two dimensional data (e.g. spreadsheets and tables). `AccessibleExtendedTable` provides a few additional methods that are needed by most accessibility uses of tables, and so should always be implemented by any user interface component that implements `AccessibleTable`.

Presented in logical groups, these are the methods of the `AccessibleTable` and the `AccessibleExtendedTable` interfaces:

row/column orientation methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(d)(2)):

— `getAccessibleAt()`

— `getAccessibleColumn()`

— `getAccessibleColumnExtentAt()`

— `getAccessibleIndex()`

— `getAccessibleRow()`

— `getAccessibleRowExtentAt()`

selection-related methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(a)):

— `isAccessibleColumnSelected()`

— `isAccessibleRowSelected()`

— `isAccessibleSelected()`

— `getSelectedAccessibleColumns()`

— `getSelectedAccessibleRows()`

row/column header related methods (which support some of the requirements in ISO/IEC 13066-1, subclause 7.1.7(d)(2)):

— `getAccessibleColumnDescription()`

— `getAccessibleColumnHeader()`

— `getAccessibleRowDescription()`

— `getAccessibleRowHeader()`

— `setAccessibleColumnDescription()`

— `setAccessibleColumnHeader()`

— `setAccessibleRowDescription()`

— `setAccessibleRowHeader()`

general methods relating to the table overall:

— `getAccessibleCaption()`

— `getAccessibleColumnCount()`

— `getAccessibleRowCount()`

— `getAccessibleSummary()`

— `setAccessibleCaption()`

— `setAccessibleSummary()`

4.2.8 The AccessibleText, AccessibleEditableText, AccessibleExtendedText, and AccessibleHypertextText interfaces

The `AccessibleText` and `AccessibleExtendedText` optional interfaces should be implemented by those user interface components that present more than just a few words of text (a user interface component that is a button with the text “OK” in it doesn't need to implement this interface). As with `AccessibleExtendedTable` above, both interfaces should always be implemented together. Additionally, where the text is editable by the user, the `AccessibleEditableText` optional interface should be implemented. Finally, where the text is hypertext, the `AccessibleHyperText` optional interface should be implemented.

These are the methods of the `AccessibleText` and `AccessibleExtendedText` interfaces (several of which support the requirements in ISO/IEC 13066-1, subclause 7.1.7(c)):

— `getAfterIndex()`

— `getAtIndex()`

— `getBeforeIndex()`

— `getCaretPosition()`

— `getCharacterAttribute()`

— `getCharacterBounds()`

— `getCharCont()`

— `getIndexAtPoint()`

— `getSelectedText()`

— `getSelectionEnd()`

— `getSelectionStart()`

— `getTextBounds()`

— `getTextRange()`

— `getTextSequenceAfter()`

— `getTextSequenceAt()`

— `getTextSequenceBefore()`

These are the methods of the `AccessibleEditableText` interface, which enable programmatic editing of text:

— `cut()`

— `delete()`

— `getTextRange()`

— `insertTextAtIndex()`

— `paste()`

— `replaceText()`

— `selectText()`

— `setAttributes()`

— `setTextContents()`

These are the methods of the `AccessibleHyperText` interface, which support hyperlinks in text:

— `getLink()`

— `getLinkCount()`

— `getLinkIndex()`

4.2.9 The `AccessibleValue` interface

The `AccessibleValue` optional interface should be implemented by those user interface components that have an underlying numerical value that can be changed within a range (e.g. a scrollbar or slider). These are the methods of the `AccessibleValue` interface (which support the requirements in ISO/IEC 13066-1, subclause 7.1.7(b)):

— `getCurrentAccessibleValue()`

— `getMaximumAccessibleValue()`

— `getMinimumAccessibleValue()`

— `setCurrentAccessibleValue()`

4.3 Implementing the Java accessibility API

The three subclauses below describe what a developer must do to implement the Java accessibility API – using user interface components that already implement the Java accessibility API, when developing custom user interface components that subclass components that already implement the Java accessibility API, and when creating user interface components “from scratch”. Everything in 4.3.1 must be done with any user interface component – so it applies in all cases. Likewise, it is common when creating components “from scratch”, that some of those will subclass others, so everything in 4.3.2 also applies to 4.3.3.

4.3.1 Using existing accessible user interface components

When using existing accessible user interface components, it is critical that correct accessibility metadata be added where needed. The places for accessibility metadata are:

— the `AccessibleName`, and where appropriate, the `AccessibleDescription` of a user interface component

— the descriptions of images/icons

— relationships between components

In addition to these metadata cases, there is also the exceedingly rare case where the developer needs to alter the apparent user interface component hierarchy – explicitly setting the parent of of a user interface element to something other than the default.

4.3.1.1 Setting the AccessibleName and AccessibleDescription

It is critical that every user interface component that a user might in some way interact with have an `AccessibleName`. This is the text that a screen reader will speak or braille, or a voice recognition command-and-control system would put into its recognition grammar, or an on-screen keyboard would display on its dynamic keyboard. Many user interface components (such as those in the `javax.swing.*` package) will use whatever appropriate text they can as a default `AccessibleName`. For example, the text displayed on a `javax.swing.JButton` is returned as the default `AccessibleName`. Thus, so long as the button has text, it has a name. Where there is no such text, it must be set explicitly by the application. This is done as follows:

```
// Example of setting the AccessibleName of a Swing component
myTextedButton = new javax.swing.JButton("OK");
// no more need be done - "OK" used by default
...
myNonTextButton = new javax.swing.JButton();  // no text set
myNonTextButton.getAccessibleContext().setAccessibleName
 ("Button name");
```

For many user interface components it is often useful to have an `AccessibleDescription`. This is the text that a screen reader will speak or braille if a user asks for more information about the user interface element. It may also be helpful for people with cognitive impairments. Many user interface components (such as those in the `javax.swing.*` package) will use the tooltip text as the `AccessibleDescription`. Where there is no such text, it must be set explicitly by the application. This is done as follows:

```
// Example of setting the AccessibleDescription of a Swing
// component
myToolTippedComponent = new javax.swing.JComponent();
myToolTippedComponent.setToolTipText("Displays page preview");
// no more need be done - "Displays page preview" used by default
...
myPlainComponent = new javax.swing.JComponent();
myPlainComponent.getAccessibleContext().setAccessibleDescription
 ("Shows the page layout");
```

Because the `AccessibleName` and `AccessibleDescription` are “user visible” strings, they must be localized to the language of the user interface.

4.3.1.2 Setting Accessible Relationships

Many developers design applications where user interface components have visibly obvious relationships. The canonical example is that of a set of labeled text fields, where the user navigates through the fields, entering the information prompted for (name, address, phone number, etc.). However, unless the visibly obvious relationship is formally encoded in some fashion, assistive technologies will have to either guess or otherwise fail to convey this to the user. The screen reader would simply speak “Edit Text... Edit Text... Edit Text” as the user Tabbed through the fields; the speech recognition control system couldn't put “Name” and “Address” and “Phone” into its recognition grammar and have the ability to bring the focus to the associated field.

And beyond labeling relationships such as the text entry field and its prompt or label, there are a variety of other useful kinds of relationships that should be encoded for assistive technologies. Furthermore, in the Java accessibility API most relationships are encoded in pairs – each of the two user interface components in a relationship generally reference the other(s) they are related to.

The most important relationships defined in the Java accessibility API are:

— `CONTROLLER_FOR` / `CONTROLLED_BY` – used to denote when one user interface component manipulates another, such as in a spreadsheet when the value of one cell controls the value of another cell (e.g. the *First Quarter Sales* number is a `CONTROLLER_FOR` the *First Quarter Net Revenue* number)

— `LABEL_FOR` / `LABELED_BY` – used to denote when one user interface component is the label for another, such as text fields and their prompts or labels

— `MEMBER_OF` – used to denote when one user interface component is a member of a group, such as a radio button that is part of a radio button group – but also useful in a spreadsheet where one cell is part of a group of cells (e.g. the *First Quarter Sales* number is a `MEMBER_OF` the text *First Quarter* field/column of the table and also a `MEMBER_OF` the text *Sales* field/row of the table)

As relationship names are simply text keys in a (key, target) pair in the `AccessibleRelation` class, new keys can be defined – and they will be provided to assistive technologies which request them. Care should be taken when doing this, however – unless there is broad understanding of the meaning of any such new relationship key text, assistive technologies won't know what to do with it and the meaning will be lost.

Setting up accessible relationships between components is done as follows:

```
// Example of setting up Accessible Relationships between
// components
myControllerComponent = new javax.swing.JComponent();
myControlledComponent = new javax.swing.JComponent();
myControllerComponent.getAccessibleContext().
  getAccessibleRelationSet().add(newAccessibleRelation
 (javax.accessibility.AccessibleRelation.CONTROLLER_FOR,
myControlledComponent.getAccessibleContext());
myControlledComponent.getAccessibleContext().
 getAccessibleRelationSet().add(newAccessibleRelation
 (javax.accessibility.AccessibleRelation.CONTROLLED_BY,
myControllerComponent.getAccessibleContext());
```

This is the same pattern for all relationships. Note however that the Swing library has a simpler convenience method for the most common relationship: the label relationship. The `javax.swing.JLabel` includes the method `setLabelFor()`. This method takes as its argument another `javax.swing.JComponent` and when used in that fashion, will also set up the pair of accessible labeling relationships. This is done as follows:

```
// Example of setting up a labeling relationships using the
// Swing JLabel
myNameLabel = new javax.swing.JLabel("Name:");
myNameTextEntry = new  javax.swing.JTextField();
myNameLabel.setLabelFor(myNameTextEntry);
```

4.3.1.3 Setting the Description of an Image or Icon

It is sometimes appropriate to set the description of images and icons in the user interface. These descriptions are distinct from the `AccessibleName` and `AccessibleDescription` of the user interface component containing the image or icon (e.g. a button without any text but displaying the image of a trash can). The image description is specifically that – a description of the image itself. The context and specific use of the image may differ, but the fact that the same image is used indicates that a similar concept is at play. A screen reader user may have the image/icon description spoken or brailled to them when they specifically ask for more information from their screen reader. This is done as follows:

```
// Example of setting up the image description of an image
myImage = new javax.swing.ImageIcon(new java.net.URL("file:///trash.gif");
myImage.getAccessibleContext().getAccessibleIcon()[0].
 getAccessibleIconDescription("Trash can with lid beside it");
// the above assumes only one image for the ImageIcon...
```

4.3.1.4 Setting the AccessibleParent

In certain rare cases it is necessary to formally set a user interface component hierarchy that is different from what is derived from the base component tree. The most common of this unlikely case is when creating what are called fly-weight components – components that don't have the full machinery of the base user interface component class of the library. For example, the `javax.swing.JTabbedPane` component contains a number of fly-weight tabs. These tabs don't extend `javax.swing.JComponent`. Therefore the accessibility API implementation for the fly-weight tab (`javax.swing.JTabbedPane$Tab.AccessibleTab`) must formally set its `AccessibleParent`.

Understanding precisely when this should be done is beyond the scope of this document. A careful study of the accessibility API implementation of the Swing `JTabbedPane` is recommended for developers who may be in this situation.

4.3.2 Subclassing existing accessible user interface components

To create a custom accessible user interface component using an existing accessible component (e.g. from the `javax.swing.*` package), choose the existing component that most closely matches the custom component ("subclassing high"). For example, to create a custom button, start with `javax.swing.JButton`. If the accessibility API implementation is the same (nothing unusual is occurring in the custom component), then this is sufficient:

```
// Example of "subclassing high" - with unchanged accessibility
class CustomButton extends javax.swing.JButton {
   // custom code here
}
```

If the custom button will have – for example – a different `AccessibleRole`, then additional code is required: the call `getAccessibleRole()` will need to return the correct role. This is accomplished by overriding that method call in the `AccessibleContext` returned by `CustomButton.getAccessibleContext()`:

```
// Example of "subclassing high" - with custom accessibility
class CustomButton extends javax.swing.JButton {
    // custom code here
    public AccessibleContext getAccessibleContext() {
        if (accessibleContext == null) {
            accessibleContext = new AccessibleCustomButton();
        }
    }
    return accessibleContext;
    class AccessibleCustomButton extends AccessibleJButton {
        public AccessibleRole getAccessibleRole() {
            return "CustomRole";
        }
    }
}
```

If more customization is done, relatively more work will need to be done in the subclass.

On the other hand, if the custom component is completely unlike an existing Swing component – but still otherwise leverages the Swing framework – then that component would subclass `javax.swing.JComponent` ("subclassing low"):

```
// Example of "subclassing low" - will need to implement
// accessibility
class CustomComponent extends javax.swing.JComponent {
    // custom code here
}
```

In such a case, the `CustomComponent` must create an `AccessibleContext` class, and implement all of the accessibility API as appropriate to that custom component.

In either case ("high" or "low"), the use of this custom component must also follow the guidance in subclause 14.

4.3.3 Creating accessible user interface components "from scratch"

The Java accessibility API supports use with user interface components created "from scratch" (e.g. which aren't subclassing one or more accessible components such as those in the `javax.swing.*` package.) When creating accessibility support from scratch, the developer will need to implement the entirety of the accessibility API for each user interface component – as appropriate for that component (what constitutes "appropriate" is the bulk of this document). The recommended way of doing this is to leverage an object-oriented user interface component system, implementing base accessibility API support on a base user interface component class, and the overriding and additional support where appropriate in the subclasses. A good example of this is the implementation in the Swing classes – whose source code may be studied in the aid of such an implementation (and further, may be outright copied directly by code that either follows the LGPL license of the OpenJDK, or obtains a commercial license from Oracle).

4.3.3.1 Using Swing as a model – inner classes

For the sake of code maintenance, the accessibility API implementation in Swing is done through the use of inner classes. This keeps the implementation code in the same source file of the component, while not cluttering the component's public API.

The core of the implementation for Swing is in the `javax.swing.$AccessibleJComponent` class. `AccessibleJComponent` is a subclass of `AccessibleContext`, and implements the `AccessibleComponent`, and `AccessibleExtendedComponent` interfaces. This base implementation not only provides default implementations of all of the `AccessibleContext`, `AccessibleComponent`, and `AccessibleExtendedComponent` methods; it also implements comment event tracking for accessibility `PropertyChange` events. However, since the `javax.swing.JComponent` class should never be instantiated directly, `javax.swing.JComponent$AccessibleJComponent` is abstract - it needs to be subclassed in order to be used. Specifically, any subclass should at a minimum implement `javax.accessibility.getAccessibleRole()`.

Then each `JComponent` subclass likewise contains an inner class that in turn subclasses `JComponent$AccessibleJComponent`. And, where appropriate, adds additional API functionality. For example, the javax.swing.AbstractButton class is the superclass of not only the `javax.swing.JButton` class, but also `javax.swing.JMenuItem` and `javax.swing.JToggleButton`. As all of these can be programmatically "pressed", the inner class `javax.swing.AbstractButton$AccessibleAbstractButton` class not only subclasses `JComponent$AccessibleJComponent`, it also implements the interface `javax.accessibility.AccessibleAction`. This shared inner class implementation is then re-used by `JButton`, `JMenuItem` and `JToggleButton`.

As noted above, Swing has flyweight components – specifically `javax.accessibility.JTabbedPane$Tab`. Since it is a flyweight, it doesn't extend `JComponent` and so a full (and essentially duplicate) implementation of `JComponent$AccessibleJComponent`, must be contained within `JTabbedPane$Tab$AccessibleTab`.

5 Exposing User Interface Element Information

In ISO/IEC 13066-1, subclause 7.1.7 requires that applications

> *provide AT with information about user interface elements, including but not limited to:*

a) role, state(s), boundary, name, and description of the user interface element

b) current value and any minimum or maximum values, if the user interface element represents one of a range of values

c) text contents, text attributes, and the boundary of text rendered to the screen.

d) the relationship of the user interface element to other user interface elements

   1) in a single data value, whether this user interface element is a label for another user interface element or is labelled by another user interface element

   2) in a table, the row and column that it is in, including headers of the row and column if present

   3) in a hierarchical relationship, any parent containing the user interface element, and any children contained by the user interface element

Subclauses 5.1 through 5.4 describe how the Java accessibility API supports each of the requirements in subclause 7.1.7.

5.1 Role, state(s), boundary, name, and description of the user interface element

Accessible role and state(s) information, as well as the accessible name and description of a user interface component, is part of the `AccessibleContext` class, required of every accessible user interface component. Boundary information of a user interface component is part of the `AccessibleComponent` optional interface, which must be implemented by every user interface component that is drawn to the screen.

Specifically:

5.1.1 Role information

Assistive technologies obtain the role of a user interface component by first obtaining that component's `AccessibleContext`, and from that, the `AccessibleRole`. For purposes of presenting this role to a user (e.g. to speak the role name via a speech synthesizer), an assistive technology may obtain a text String from the `AccessibleRole` (which by default will be in the `Locale` of the application, though other `Locales` may be explicitly obtained – e.g. to obtain a French human-readable role `String` from an application that is otherwise localized to German):

```
// Examples of getting the AccessibleRole - default and French
AccessibleRole r =
 myComponent.getAccessibleContext().getAccessibleRole();
String defaultRoleText = r.toDisplayString();
String frenchRoleText = r.toDisplayString(Locale.FR);
```

For applications that are creating custom components, it will generally be necessary to explicitly set the `AccessibleRole` for that custom component. This is typically done by subclassing – either subclassing an existing specific accessible user interface component (e.g. creating a specialized button component by subclassing the Swing `javax.swing.JButton` component), or creating a custom component from a base class like Swing `javax.swing.JComponent`), or creating it entirely from scratch. In the subclassing case, the accessibility implementation must be subclassed, with the implementation subclass providing the appropriate `AccessibleRole`: This is only necessary if the subclass is a different kind of object than the subclassed class (e.g. creating something that isn't a button when subclassing Swing `JButton`), or when subclassing a base class like `JComponent`:

```
// Example of setting the AccessibleRole of a custom component
class CustomButton extends javax.swing.JComponent {
    class AccesibleCustomButton extends
      javax.swing.JComponent.AccessibleJComponent {
        protected AccessibleRole getAccessibleRole() {
            return AccessibleRole.PUSH_BUTTON;
        }
    }
}
```

5.1.2 State(s) information

Assistive technologies obtain the state(s) of a user interface component by first obtaining that component's `AccessibleContext`, and from that, the `AccessibleStateSet` which contains one or more specific `AccessibleStates`. For purposes of presenting these states to a user (e.g. to speak the names of the states via a speech synthesizer), an assistive technology may obtain a text String from each of the the `AccessibleStates` (which by default will be in the Locale of the application, though other Locales may be explicitly obtained – e.g. to obtain a French human-readable role `String` from an application that is otherwise localized to German):

```
// Example of getting the AccessibleStates - default and French
AccessibleStateSet states =
 myComponent.getAccessibleContext().getAccessibleStateSet();
String defaultStatesText = states.toDisplayString();
AccessibleState statesArray[] = states.toArray();
String defaultFirstStateText = statesArray[0].toDisplayString();
String frenchFirstStateText =
 statesArray[0].toDisplayString(Locale.FR);
```

If an assistive technology wants to check explicitly for a specific state, it can do so:

```
// Example of getting a specific AccessibleState
AccessibleStateSet states =
 myComponent.getAccessibleContext().getAccessibleStateSet();
if (states.contains(AccessibleState.CHECKED) {
    // AT logic for handling a component that is checked
}
```

For applications that are creating custom components, it will generally be necessary to explicitly set the `AccessibleStates` for that custom component. This is typically done by setting the states as through the listener pattern on the component being made accessible, or within the component's or application's code.

```
// Example of updating a specific AccessibleState based on the
// listener pattern
class CustomButton extends javax.swing.JComponent {
    class AccessibleCustomButton extends
      javax.swing.JComponent.AccessibleJComponent {
        protected class AccesibleCustomButton() {
            super();
            protected class StateHandler implements
             PropertyChangeListener {
                public void propertyChange
                 (ProptertyChangeEvent evt) {
                    if (evt.getPropertyName() == "some name" &&
                        evt.getNewValue() == "some value") {
                        AccesibleCustomButton.this.states.add
                        ("some state");
                }
            }
        }
        public AccessibleStateSet getAccesibleStateSet() {
            AccessibleStateSet states =
             super.getAccessibleStateSet();
            // state explicitly added below
            states.add(AccessibleStateSet.OPAQUE);
        }
    }
}
```

5.1.3 Boundary information

Assistive technologies obtain the bounding rectangle of a user interface component by first obtaining that component's `AccessibleContext`, and from that, the `AccessibleComponent`, and finally from that the boundary information. An assistive technology can obtain this in a variety of ways – it can obtain just the position of the user interface component (in local or screen coordinates), or just its size, or its bounding rectangle (in both local coordinates):

```
// Example of getting the boundary of a component
AccessibleComponent myAccessibleComponent =
 myComponent.getAccessibleContext().getAccessibleComponent();
Point localLocation = myAccessibleComponent.getLocation();
Point screenLocation = myAccessibleComponent.getLocationOnScreen();
Dimension size = myAccessibleComponent.getSize();
Rectangle localRectangle = myAccessibleComponent.getBounds();
```

5.1.4 Name information

Assistive technologies obtain the name of a user interface component by first obtaining that component's `AccessibleContext`, and from that, the name. The name is always a human-readable string, which may be presented directly to a user via the assistive technology (e.g. to speak the name via a speech synthesizer). It is always in the `Locale` of the application:

```
// Example of getting the AccessibleName of a component
String name =
 myComponent.getAccessibleContext().getAccessibleName();
```

For applications that are creating custom components, it will generally be necessary to explicitly set the accessible name for that custom component. This is typically done either in the subclass itself, or simply by explicitly setting the accessible name from within the application:

```
// Example #1 of setting the AccessibleName of a component
class CustomButton extends javax.swing.JComponent {
    class AccessibleCustomButton extends
      javax.swing.JComponent.AccessibleJComponent {
        protected String getAccessibleName() {
            // assumes such a method below
            return CustomButton.this.getButtonText();
        }
    }
}
```

or

```
// Example #2 of setting the AccessibleName of a component
AccessibleContext getAccessibleContext() {
    AccessibleContext ac = new AccessibleCustomButton();
    ac.setAccessibleName("some button name");
    return ac;
}
```

5.1.5 Description information

Assistive technologies obtain the description of a user interface component by first obtaining that component's `AccessibleContext`, and from that, the description. The description is always a human-readable string, which may be presented directly to a user via the assistive technology (e.g. to speak the name via a speech synthesizer). It is always in the `Locale` of the application:

```
// Example of getting the AccessibleDescription of a component
String description =
 myComponent.getAccessibleContext().getAccessibleDescription();
```

For applications that are creating custom components, it may be appropriate to explicitly set the accessible description for that custom component (not every user interface component needs a description). This is typically done either in the subclass itself, or simply by explicitly setting the accessible description from within the application:

```
// Example #1 of setting the AccessibleDescription of a component
class CustomButton extends javax.swing.JComponent {
    class AccessibleCustomButton extends
      javax.swing.JComponent.AccessibleJComponent {
        protected String getAccessibleDescription() {
            // assumes such a method below
            return CustomButton.this.getButtonToolTip();
        }
    }
}
```

or

```
// Example #2 of setting the AccessibleDescription of a component
AccessibleContext getAccessibleContext() {
    AccessibleContext ac = new AccessibleCustomButton();
    ac.setAccessibleDescriptoin("some button description");
    return ac;
}
```

5.2 Current value and any minimum or maximum values, if the user interface element represents one of a range of values

Accessible value information – including the minimum & maximum values, are contained in the optional interface `AccessibleValue`. This must be implemented by all user interface components that take on one of a range of values.

Assistive technologies obtain value information by first verifying that the user interface component has value information to provide, and then by querying that information. Assistive technologies can also register a listener for value changes, and thus be automatically informed any time the value of a user interface component changes:

```
// Example of getting the current, min, and max AccessibleValues
// of a component
AccessibleValue myValue =
 myComponent.getAccessibleContext().getAccessibleValue();
if (myValue != null) {
    Number minimum = myValue.getMinimumAccessibleValue();
    Number maximum = myValue.getMaximumAccessibleValue();
    Number current = myValue.getCurrentAccessibletValue();
}
```

and/or

```
// Example of tracking changes in the AccessibleValue of a
// component
protected class ValueHandler implements PropertyChangeListener {
    public void propertyChange (ProptertyChangeEvent evt) {
      if (evt.getPropertyName() == "AccessibleValueProperty") {
            Number newValue = (Number) eve.getNewValue();
        }
    }
}
```

5.2.1 Additional value information: setting values

As noted below in subclause 34, the Java Accessibility API also supports programmatic setting of values of user interface components. This is supported not only for applications which may utilize this functionality for custom components, but also for assistive technologies. With this support, assistive technologies such as voice recognition command-and-control applications, as well as sophisticated on-screen keyboards – can explicitly set the values of sliders and scroll bars, and any other user interface component that takes on one of a range of values. This is done as follows:

```
// Example of setting the AccessibleValue of a component
AccessibleValue myValue =
 myCompnent.getAccessibleContext().getAccesibleValue();
if (myValue != null) {
    myValue.setCurrentAccessibleValue(5);   // sets new value to 5
}
```

5.3 Text contents, text attributes, and the boundary of text rendered to the screen

Accessible text information – whether for static text or editable text – is contained in the `AccessibleText` optional interface. Accessible information about editable text is contained in the `AccessibleEditableText` optional interface (a formal sub-interface of `AccessibleText`). Some additional information is available through the `AccessibleExtendedText` optional interface (the other formal sub-interface of `AccessibleText`).

Text contents can be obtained a variety of ways:

— the character/word/sentence that is located at/before/after a particular text offset or index - using the `AccessibleText.getAfterIndex()`, `AccessibleText.getAtIndex()`, and `AccessibleText.getBeforeIndex()` methods

— the text that is currently selected - using the `AccessibleText.getSelectionStart()`, `AccessibleText.getSelectionEnd()`, and `AccessibleText.setSelectedText()` methods

— the range of text between two indicies - using either the `AccessibleEditableText.getTextRange()`, or `AccessibleExtendedText.getTextRange()` methods

— the line of text containing a given index (as that line is displayed/broken on the screen, including any word-wrap used to break the line to fit in an adjustable width rectangle) – or line prior or after the line containing the given index - using the `AccessibleExtendedText.getTextSequenceAfter(AccessibleExtendedText.LINE)`, `AccessibleExtendedText.getTextSequenceAt(AccessibleExtendedText.LINE)`, and `AccessibleExtendedText.getTextSequenceBefore(AccessibleExtendedText.LINE)` methods

— the text containing a contiguous run of attributes (allowing an assistive technology to rapidly iterate through runs of text attributes, typically to efficiently display attributes on a refreshible braille display or for attribute-based searches) - using the `AccessibleExtendedText.getTextSequenceAfter(AccessibleExtendedText.ATTRIBUTE_RUN)`, `AccessibleExtendedText.getTextSequenceAt(AccessibleExtendedText.ATTRIBUTE_RUN)`, and `AccessibleExtendedText.getTextSequenceBefore(AccessibleExtendedText.ATTRIBUTE_RUN)` methods

Text attribute information can be obtained in two different ways:

— the character attributes for the single character at a particular text offset or index - using the `AccessibleText.getCharacterAttribute()` method

— the character attributes for the characters in a contiguous run of characters sharing the same set of attributes at a particular text offset or index - using the `AccessibleExtendedText.getTextSequenceAfter(AccessibleExtendedText.ATTRIBUTE_RUN)`, `AccessibleExtendedText.getTextSequenceAt(AccessibleExtendedText.ATTRIBUTE_RUN)`, and `AccessibleExtendedText.getTextSequenceBefore(AccessibleExtendedText.ATTRIBUTE_RUN)` methods

Text boundary information can be obtained in two different ways:

— the bounding rectangle of the single character at a particular text offset or index - using the `AccessibleText.getCharacterBounds()` method

— the bounding rectangle for the range of characters in a contiguous run - using the `AccessibleExtendedText.getTextBounds()` method

As described below in subclause 34, the Java accessibility API provides several API calls for making modifications to text contents.

Finally, there is another method in these accessible text interfaces, which provide functionality going beyond what is required in ISO/IEC 13066-1: an API call to count of the number of characters in a text user interface component.

There is also a fourth accessible text interface in the Java accessibility API - `AccessibleHypertext` - which provides a way for assistive technologies to locate and use hyperlinks that may be embedded in hypertext.

5.4 The relationship of the user interface element to other user interface elements

Information about the pixel location of user interface elements is part of the `AccessibleComponent` optional interface, required of every accessible user interface component showing on the screen. This has already been described in subclause 20, earlier in this clause.

Other information about the relationship of one user interface element to others is described below, in support of the requirements in subclause 7.1.7, item d.

5.4.1 in a single data value, whether this user interface element is a label for another user interface element or is labelled by another user interface element

Information about whether a user interface component is a label or not is contained in the `AccessibleRole`, and retrieval of that information is described in subclause 19 above. Labels generally have the `AccessibleRole` of `AccessibleRole.LABEL`. However, more important than any role is whether in fact the user interface component is in a labeling relationship with another user interface component. To ascertain this, an assistive technology must obtain the set of `AccessibleRelations` for a given user interface component, and see if one of those is the `AccessibleRelation.Label_For` relation (and which user interface component is being labeled). Conversely, any labeled user interface component should have a `AccessibleRelation.Labeled_By` relation, referencing the label in question. As with `AccessibleStates`, a user interface component can have multiple of them, and they are returned in an an `AccessibleStateSet`. The following code illustrates this:

```
// Example #1 of ascertaining whether a labeling relationship
// exists
AccessibleRelationSet relations =
 myComponent.getAccessibleContext().getAccessibleRelationSet();
if (relations.contans(AccessibleRelation.LABEL_FOR)) {
    // myComponent is a label!
    AccessibleRelation labelRelation =
     relations.get(AccessibleRelation.LABEL_FOR);
    AccessibleContext labeledAccesible =
     (AccessibleContext) labelRelation.getTarget();
}
```

and conversely

```
// Example #2 of ascertaining whether a labeling relationship
// exists
AccessibleRelationSet relations =
 myComponent.getAccessibleContext().getAccessibleRelationSet();
if (relations.contans(AccessibleRelation.LABELED_BY)) {
    // myComponent is a label!
    AccessibleRelation labelerRelation =
     relations.get(AccessibleRelation.LABEL_FOR);
    AccessibleContext labelerAccesible =
     (AccessibleContext) labelRelation.getTarget();
}
```

5.4.2 in a table, the row and column that it is in, including headers of the row and column if present.

Information about whether a user interface component is a data element within a data table is contained in the user interface hierarchy of the component – whether the parent user interface component implements the `AccessibleTable` interface. In some cases the user interface component may be part of a hierarchy of user interface components, in which case it won't be the immediate parent, but a grandparent. The following code shows how to determine whether a user interface component is directly a child of a table. More common would be to traverse down a user interface component hierarchy, and into a data element through the `AccessibleTable` interface. Once an assistive technology has a reference to the `AccessibleTable` interface, it is a simple matter to obtain the row & column information, and the headers of the row and column if present. These too are illustrated in the code sample below:

```
// Example of ascertaining whether a component is in a table,
// and where
AccessibleContext parent =
 myComponent.getAccessibleContext().getAccessibleParent();
if ((AccessibleTable table = parent.getAccessibleTable()) != null
 && table instanceof AccessibleExtendedTable) {
    // we know that myComponent is part of the AccessibleTable
    // 'table' and we get the extended table info for row/column
    // calculations
    int index =
     myComponent.getAccessibleContext().
     getAccessibleIndexInParent();
    int row = ((AccessibleExtendedTable) table).
     getAccessibleRow(index);
    int column = ((AccessibleExtendedTable) table).
     getAccessibleColumn(index);
    AccessibleTable rowHeaders = table.getAccessibleRowHeader();
    // assumes one header per row & column - could have multiple
    // levels
    AccessibleContext rowHeader =
     rowHeaders.getAccessibleAt(row, 0);
    AccessibleContext columnHeader =
     rowHeaders.getAccessibleAt(0, column);
}
```

5.4.3 in a hierarchical relationship, any parent containing the user interface element, and any children contained by the user interface element

Parent/child hierarchical relationship information is part of the core `AccessibleContext` class implemented by all accessible user interface components. The following code example illustrates this:

```
// Example of getting the parent of a component
AccessibleContext parent =
 myComponent.getAccessibleContext().getAccessibleParent();
```

and conversely

```
// Example of getting the children of a component
int children_count =
 myComponent.getAccessibleContext().getAccessibleChildrenCount();
AccessibleContext child;
while (int n=0; n < children_count; n++) {
    child =
     myComponent.getAccessibleContext().getAccessibleChild(n);
}
```

In addition, in certain very rare and special circumstances this “default” or “UI hierarchy” information can be overridden using the `AccessibleRelation` interface. This very rare situation arises when the underlying user interface toolkit has an illogical heirarchy that otherwise cannot be changed (which is something that can usually be done explicitly by applications which can formally change the accessibility UI heirarchy as distinct from the component heirarchy – see Setting the Accessible Parent in subclause 4.3.1.4). In that case, the parent user interface component will have an `AccessibleRelation.PARENT_WINDOW_OF` relation with each of is children, and conversely each child will have an `AccessibleRelation.SUBWINDOW_OF` with each of its parents. The usual relation calls (shown in the code sample above in subclause 24) are used to obtain this information.

5.4.3.1 in other, more generalized relationships

The Java accessibility API provides a very rich and flexible means of encoding all kinds of relationships between user interface components, going beyond what is required by ISO/IEC 13066-1. The full set of pre-defined relations is documented in the `AccessibleRelation` javadoc. These relations include:

— a set of text-flow relations, allowing assistive technologies to follow the flow of text broken up among multiple user interface components – and across multiple columns or skipping over headers/footers as displayed in a typical word processor

— a set of embedding relations, allowing assistive technologies to recognize when a "subworld" is embedded within a frame (such as is common with technologies like Object Linking and Embedding)

— a set of grouping relations, indicating when one or more user interface components are a member of a common group (and what the grouping user interface component is)

— a set of controlling/controlled relations, indicating when one or more user interface components impact or control the result of another – such as in the case of a spreadsheet when one cell is the result of a formula of other cells

— an overriding the typical UI hierarchy, to note that one object is really the child of another (used in certain UI components displaying trees, where the immediate UI hierarchy is only one level deep – a flat set of children – but is displayed as a tree hierarchy).

Beyond the pre-defined set of relations, arbitrary new relations can be created. However, great care must be taken when doing so, as assistive technologies will not understand the semantics of any such new relations, unless they are well documented and support for using them is implemented in assistive technologies.

6 Exposing User Interface Element Actions

In ISO/IEC 13066-1, subclause 7.1.8 requires software to:

> *programmatically expose a list of available actions on a user interface element and allow assistive technology to programmatically execute any of those actions.*

The `AccessibleAction` interface provides a method for determining the number of actions exposed by a user interface component, the description of each action, and a way to programmatically perform each of these actions. The API calls are described above in subclause 4.2.2.

Actions are used to support a variety of assistive technologies that provide user input alternatives. This includes things like on-screen keyboards and voice command and control systems. By enumerating the actions available on the user interface components in the hierarchy of a running application (whether or not they are visible), an assistive technology can re-present those components on an on screen keyboard, or can build a dynamic voice recognition grammar of the nouns of the user interface (the `AccessibleNames` of the buttons and checkboxes and so on) and the verbs (the `AccessibleActionDescriptions`); potentially adding adjectives to the grammar if needed (the `AccessibleRoles`).

Common actions are `CLICK` (the equivalent of clicking the mouse), `INCREMENTing` or `DECREMENTing` a value, `TOGGLEing` a `POPUP`. And `TOGGLEing` an `EXPANDable` object (such as a tree node).

The following sample code illustrates how an assistive technology would discover and then perform the CLICK action on a user interface component such as a button (equivalent to the user actually moving the mouse over the component and then pressing and releasing the left mouse button):

```
// Example of performing the CLICK action on a component
AccessibleAction action =
 myComponent.getAccessibleContext().getAccessibleAction();
if (action != null) {
    int count = action.getAccessibleActionCount();
    while (n = 0; n < count; n++) {
        if (action.getAccessibleActionDescription(n) ==
         AccessibleAction.CLICK) {
            action.doAccessibleAction(n);
        }
    }
}
```

In addition, the `AccessibleSelection` optional interface provides a method for programmatically adding/removing items from a container in which one or more user interface components may be selected – such as a list box or a menu. In other accessibility APIs, selection addition/removal might be part of their action interface. The API calls for the `AccessibleSelection` optional interface are described above in subclause 10.

As with actions, assistive technologies can enumerate the user interface components which provide the `AccessibleSelection` interface in the hierarchy of a running application (whether or not they are visible), and then re-present those on an on-screen keyboard or use them in a dynamic voice recognition grammar.

The following sample code illustrates how an assistive technology would select the third item (and ensure that only the third item is selected) among the children of a user interface component that supported the `AccessibleSelection` interface – such as a ListBox.

```
// Example of selecting an item in a component
AccessibleSelection selection =
 myComponent.getAccessibleContext().getAccessibleSelection();
if (selection != null) {
    selection.clearAccessibleSelection();
    selection.addAccessibleSelection(2);    // 0 is the first item
}
```

7 Keyboard Focus

In ISO/IEC 13066-1, subclause 7.1.9 requires software to:

> *programmatically expose information necessary to track and modify: focus, text insertion point (where applicable), and selection attributes of user interface elements.*

Tracking keyboard focus – whether the formal “focus” on a user interface component like a button, the “soft focus” of the choice of a menu item within a menu, the text insertion “focus”, or selection focus (whether it is selected text in an editable text field or selected items in a list box) – is simply a subset of the more general event mechanism provided by the Java accessibility API. The general event mechanism is described below in Clause 31. This clause focuses on the broader set of focus-related events.

7.1 Tracking (and modifying) focus

In the Java accessibility parlance, focus is an `AccessibleState` - specifically `AccessibleState.FOCUSED`. To determine whether a user interface component is focused, an assistive technology would examine the `AccessibleStateSet` of the component to see whether it included `AccessibleState.FOCUSED`. A code example of showing how to obtain these states is in subclause 19 above.

To track the focus, the assistive technology should add a listener to `AccessibleState PropertyChange` events, filtering those events for the state `AccessibleState.FOCUSED`. The following code example illustrates this:

```
// Example of tracking focus events from a component
protected class StateHandler implements PropertyChangeListener {
    public void propertyChange (ProptertyChangeEvent evt) {
        if (evt.getPropertyName() == AccessibleState.FOCUSED &&
            (Object newValue = event.getNewValue()) != null) {
            AccessibleContext focusedAccessible =
             (AccessibleContext) newValue;
             // 'focusedAccessible' now has focus
             // additional calls here, such as to get the
             // AccessibleName of the focusedAccessible, and
             // speak it & its role via text-to-speech
        }
    }
}
...
myComponent.getAccessibleContext().addPropertyChangeListener
 (new StateHandler());
```

Beyond tracking focus, assistive technologies can explicitly focus onto a user interface component. This is done with the `requestFocus()` call, as follows:

```
// Example of setting the focus to a component
myComponent.getAccessibleContext().requestFocus();
```

NOTE this only works for user interface components that have a “hard focus” - where the user interface explicitly defines a “focus” state or property for the component. For “soft focus” components like menu items in Swing, the AccessibleSelection interface must be used instead, as illustrated in the code example in Clause 27.

7.2 Tracking (and modifying) text insertion point

As mentioned above in subclause 12, all user interface components that provide for editable text must implement the `AccessibleEditableText` interface. This interface contains methods for obtaining the text caret location (or insertion point), and for modifying it. Also methods for selecting a region of text – which may be considered a special case of an insertion point, as any text entered will take the place of the selected text. The following code sample illustrates these methods:

```
// Example of setting the focus to a component
// for all text
text = myComponent.getAccessibleContext().getAccessibleText();
int caretPosition = text.getCaretPosition();
int selectionStart = text.getSelectionStart();
int selectionEnd = text.getSelectionEnd();
String selectedText = text.getSelectedText();

// for editable text:
if (text instanceof AccessibleEditableText) {
    // sets caret location to 4, with an “empty” selection range
    ((AccessibleEditableText) text).selectText(4, 4);
}
```

To track the text insertion point, the assistive technology should add a listener to `AccessibleState PropertyChange` events, filtering those events for `ACCESSIBLE_CARET_PROPERTY` events. The following code example illustrates this:

```
// Example of tracking property change events, filtering for caret
// changes
protected class CaretHandler implements PropertyChangeListener {
    public void propertyChange (ProptertyChangeEvent evt) {
        if (evt.getPropertyName() == ACCESSIBLE_CARET_PROPERTY) {
            int caretLocation = (int) evt.getNewValue();
        }
    }
}
...
myComponent.getAccessibleContext().addPropertyChangeListener(new CaretHandler());
```

7.2.1 Tracking (and modifying) selection attributes

There are two different notions of selection in the Java accessibility API parlance – user interface components that are selected (like items in a list box or a menu), and selected text. User interface component selection has already been discussed in Clause 27.

Text selection is part of the `AccessibleEditableText` interface, and many of the methods were already discussed above in subclause 29. Determining what text is selected is done via the `AccessibleText.getSelectionStart()` and `AccessibleText.getSelectionEnd()` methods, while obtaining the selected text itself is done with the `AccessibleText.getSelectedText()` call. Setting/modifying the selection is done with the `AccessibleEditableText.selectText()` method. Once text is selected, any text typed on the keyboard will replace the selection, or it can be replaced explicitly with the `AccessibleEditableText.replaceText()` method.

Tracking selection changes is done with event listeners, using the same pattern as other listeners in the Java accessibility API – and very much the same as tracking the caret / insertion point, as described in subclause 29 above, but by listening for `ACCESSIBLE_SELECTION_PROPERTY` changes:

```
// Example of tracking property change events, filtering for caret
// changes
protected class SelectionHandler implements PropertyChangeListener {
   public void propertyChange (ProptertyChangeEvent evt) {
      if (evt.getPropertyName() == ACCESSIBLE_SELECTION_PROPERTY) {
          // selection has changed, query for the new selection
          AccessibleText text = (AccessibleText) evt.getSource();
          int selectionStart = text.getSelectionStart();
          int selectionEnd = text.getSelectionEnd();
      }
   }
}
...
myComponent.getAccessibleContext().addPropertyChangeListener(new
SelectionHandler());
```

Beyond selection attributes, other text attributes are available via the `AccessibleText` and `AccessibleEditableText` interfaces, as described in subclause 23. In addition to obtaining those attributes, changes in those attributes can also be tracked via assistive technologies – again going beyond the minimum required by ISO/IEC 13066-1. The same event listener pattern is used for text attribute changes as was described above in subclause 29 and also earlier in this subclause: for text caret / insert point and selection. This is done by listening for property change events of property type: `ACCESSIBLE_TEXT_ATTRIBUTES_CHANGED`.

Also, in addition to caret and selection changes, assistive technologies can track the insertion and deletion of text – by listening for property change events of property type: `ACCESSIBLE_TEXT_PROPERTY`.

8 Events

In ISO/IEC 13066-1, subclause 7.1.10 requires software to:

programmatically expose notification of events relevant to user interactions, including but not limited to:

a) changes in the user interface element value;

b) changes in the name of the user interface element;

c) changes in the description of the user interface element;

d) changes in the boundary of the user interface element;

e) changes in the hierarchy of the user interface element.

The Java accessibility API uses one primary event mechanism – based on the Java Beans pattern – to signal changes in most aspects of the user interface needed by AT. This is a `PropertyChange` notification fired from the `AccessibleContext` associated with the user interface component. In a few cases other (e.g. `AccessibleTable`), specialized events may also be fired.

AT – typically via a bridge – registers a listener for events from the `AccessibleContext`, and receives events with a `propertyName`, and an `oldValue`, and a `newValue`. With this information the AT can determine what happened, or at least determine that it needs more information (for which it will then make more API calls).

8.1 changes in the user interface element value

Those user interface elements which represent one of a range of changeable values will have an `AccessibleContext` that provides the `AccessibleValue` optional interface. AT that registers for `PropertyChange` event notification from that `AccessibleContext` will receive events whose `propertyName` is `ACCESSIBLE_VALUE_PROPERTY` and whose `oldValue` is the old value and whose `newValue` is the new value.

8.2 changes in the name of the user interface element

As all user interface components must have an `AccessibleName`, all `AccessibleContext` will fire `PropertyChange` events for a change in the accessible name. AT that registers for `PropertyChange` event notification from that `AccessibleContext` will receive events whose `propertyName` is `ACCESSIBLE_NAME_PROPERTY` and whose `oldValue` is the old name and whose `newValue` is the new name.

8.3 changes in the description of the user interface element

As all user interface components must have an `AccessibleDescription`, all `AccessibleContext` will fire `PropertyChange` events for a change in the accessible name. AT that registers for `PropertyChange` event notification from that `AccessibleContext` will receive events whose `propertyName` is `ACCESSIBLE_DESCRIPTION_PROPERTY` and whose `oldValue` is the old description and whose `newValue` is the new description.

8.4 changes in the boundary of the user interface element

Those user interface elements which are drawn to the screen (even only at some potential time in the future such as an undisplayed menu item) will have an `AccessibleContext` that provide an `AccessibleComponent` optional interface. AT that registers for `PropertyChange` event notification from that `AccessibleContext` will receive events whose `propertyName` is `ACCESSIBLE_COMPONENT_BOUNDS_PROPERTY` and whose `oldValue` is the old bounding rectangle and whose `newValue` is the new bounding rectangle.

8.5 changes in the hierarchy of the user interface element

MSAA events are used to notify ATs of changes in the hierarchy of a user interface element.

8.6 changes in other accessibility aspects of user interface components

Much of the information available via the Java accessibility API can be “listened” for, and any time that information changes, a listening assistive technology will receive notification of the change. The set of properties that for which changes trigger events is documented in the javadoc for `AccessibleContext`:

— A change in the number of actions available to a user interface component that implements the `AccessibleAction` interface (expressed via `ACCESSIBLE_ACTION_PROPERTY` change events)

— A change in the “active descendant” of a user interface component that has the `AccessibleState.MANAGES_DESCENDANTS` – used in cases where the number of children may be unbounded or simply very large, where it is not practical to enumerate all of the children (expressed via `ACCESSIBLE_ACTIVE_DESCENDANT_PROPERTY` change events)

— A change in the caret location available to a user interface component that implements the `AccessibleText` interface (expressed via `ACCESSIBLE_CARET_PROPERTY` change events)

— The addition or removal of a child user interface component from its parent (expressed via `ACCESSIBLE_CHILD_PROPERTY` change events)

— A change in which portion of hypertext has focus for a user interface component that implements the `AccessibleHypertext` interface (expressed via `ACCESSIBLE_HYPERTEXT_OFFSET` change events)

— A change indicating that a significant change has occurred to the children of a user interface component – typically a list or tree or text – and indicating that assistive technology should re-build any model it has of those children (expressed via `ACCESSIBLE_INVALIDATE_CHILDREN` change events)

— A change in the accessible selection of a user interface component that implements the `AccessibleSelection` interface (expressed via `ACCESSIBLE_SELECTION_PROPERTY` change events)

— A change in the accessible states of a user interface component – e.g. a component becoming focused, or checked (expressed via `ACCESSIBLE_STATE_PROPERTY` change events)

— A change in the caption of a user interface component that implements the `AccessibleTable` interface (expressed via `ACCESSIBLE_TABLE_CAPTION_CHANGED` change events)

— A change in the description of a table column of a user interface component that implements the `AccessibleTable` interface (expressed via `ACCESSIBLE_TABLE_COLUMN_DESCRIPTION_CHANGED` change events)

— A change in the column header of a user interface component that implements the `AccessibleTable` interface (expressed via `ACCESSIBLE_TABLE_COLUMN_HEADER_CHANGED` change events)

— A change in the underlying data model of a user interface component that implements the `AccessibleTable` interface – indicating that a significant content change has occurred to the children of this table and that assistive technology should re-build any model it has of this information (expressed via `ACCESSIBLE_TABLE_MODEL_CHANGED` change events)

— A change in the description of a table row of a user interface component that implements the `AccessibleTable` interface (expressed via `ACCESSIBLE_TABLE_ROW_DESCRIPTION_CHANGED` change events)