Java Platform, Standard Edition

Internationalization Guide

Release 11
E94896-05
July 2023

ORACLE

Java Platform, Standard Edition Internationalization Guide, Release 11

E94896-05

Contents

6 Font Configuration Files

Preface

This guide summarizes the internationalization APIs and features of the Java SE Platform.

Audience

This guide is intended for Java programmers who want to design applications so that they can be adapted to various languages and regions without engineering changes.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For coding examples and step-by-step instructions, see the Internationalization Trail in The Java Tutorials (Java SE 8 and earlier).

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Internationalization Enhancements

Recent releases of the JDK include enhancements to the internationalization process to support updated standards.

Topics:

Internationalization Enhancements in JDK 11

Internationalization enhancements for JDK 11 include:

Unicode 10.0.0

Support has been added for Unicode 10.0.0. Java Platform, Standard Edition (Java SE) 9 and 10 supported Unicode 8.0.

The Unicode 10.0 standard includes 16,018 characters and 10 scripts that were introduced since Unicode 8.0, all of which are supported in Java SE 11.

Internationalization Enhancements in JDK 10

Internationalization enhancements for JDK 10 include:

Additional Unicode Language-Tag Extensions

The IETF BCP (best current practice) 47 language tags standard, which has been supported in the `Locale` class since Java SE 7, includes a Unicode extension subtag. As of Java SE 9, only the `-ca` (calendar) and `-nu` (number) extensions are supported.

Java SE 10 adds support for the following additional extensions in the relevant JDK classes:

- `-cu` (currency type)
- `-fw` (first day of week)
- `-rg` (region override)
- `-tz` (time zone)

In JDK 10, if an application specifies a locale of `en-US-u-cu-EUR`, which means US English with Euro currency, `java.util.Currency.getInstance(locale)` instantiates a Euro `Currency`. If the locale is `en-US-u-cu-JPY`, a Japanese Yen `Currency` is instantiated.

Internationalization Enhancements in JDK 9

Internationalization enhancements for Oracle Java Development Kit 9 include:

- Unicode 8.0
- CLDR Locale Data Enabled by Default
- UTF-8 Properties Files

Unicode 8.0

Support has been added for Unicode 8.0. Java Platform, Standard Edition (Java SE) 8 supported Unicode 6.2.

The Unicode 6.3, 7.0, and 8.0 standards introduced 10,555 characters, 29 scripts, and 42 blocks, all of which are supported in Java SE 9.

CLDR Locale Data Enabled by Default

The XML-based locale data of the Unicode Common Locale Data Repository (CLDR), first added in JDK 8, is the default locale data in JDK 9. In previous releases, the default was `JRE`.

There are four distinct sources for locale data, identified by the following keywords:

- `CLDR` represents the locale data provided by the Unicode CLDR project.
- `HOST` represents the current user's customization of the underlying operating system's settings. It works only with the user's default locale, and the customizable settings may vary depending on the operating system. However, primarily date, time, number, and currency formats are supported.
- `SPI` represents the locale-sensitive services implemented by the installed Service Provider Interface (SPI) providers.
- `COMPAT` (formerly called `JRE`) represents the locale data that is compatible with releases prior to JDK 9. `JRE` can still be used as the value, but `COMPAT` is preferred.

To select a locale data source, use the `java.locale.providers` system property, listing the data sources in the preferred order. If a provider cannot offer the requested locale data, the search proceeds to the next provider in order. For example:

```
java.locale.providers=HOST,SPI,CLDR,COMPAT
```

If you do not set this property, the default behavior is equivalent to the following setting:

```
java.locale.providers=CLDR,COMPAT,SPI
```

To enable behavior that is compatible with JDK 8, set the `java.locale.providers` system property to a value with `COMPAT` to the left of `CLDR`.

See the JDK 9 and JRE 9 Supported Locales page for supported locales. See java.util.spi.LocaleServiceProvider API specification for the related API.

UTF-8 Properties Files

In Java SE 9, properties files are loaded in UTF-8 encoding. In previous releases, ISO-8859-1 encoding was used for loading property resource bundles. UTF-8 is a much more convenient way to represent non-Latin characters.

Most existing properties files should not be affected: UTF-8 and ISO-8859-1 have the same encoding for ASCII characters, and human-readable non-ASCII ISO-8859-1 encoding is not valid UTF-8. If an invalid UTF-8 byte sequence is detected, the Java runtime automatically rereads the file in ISO-8859-1.

If there is an issue, consider the following options:

- Convert the properties file into UTF-8 encoding.
- Specify the runtime system property for the properties file's encoding, as in this example:

```
java.util.PropertyResourceBundle.encoding=ISO-8859-1
```

See java.util.PropertyResourceBundle.

2
Internationalization Overview

Internationalization is the process of designing an application so that it can be adapted to various languages and regions without engineering changes. Sometimes the term internationalization is abbreviated as i18n, because there are 18 letters between the first "i" and the last "n."

An internationalized program has the following characteristics:

- With the addition of localization data, the same executable can run worldwide.
- Textual elements, such as status messages and the GUI component labels, are not hardcoded in the program. Instead they are stored outside the source code and retrieved dynamically.
- Support for new languages does not require recompilation.
- Culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language.
- It can be localized quickly.

The global Internet demands global software - that is, software that can be developed independently of the countries or languages of its users, and then localized for multiple countries or regions. The Java Platform provides a rich set of APIs for developing global applications. These internationalization APIs are based on the Unicode standard and include the ability to adapt text, numbers, dates, currency, and user-defined objects to any country's conventions.

This guide summarizes the internationalization APIs and features of the Java Platform, Standard Edition. For coding examples and step-by-step instructions, see the Internationalization Trail in the Java Tutorials.

Text Representation

The Java programming language is based on the Unicode character set, and several libraries implement the Unicode standard. Unicode is an international character set standard which supports all of the major scripts of the world, as well as common technical symbols. The original Unicode specification defined characters as fixed-width 16-bit entities, but the Unicode standard has since been changed to allow for characters whose representation requires more than 16 bits. The range of legal code points is now U+0000 to U+10FFFF. An encoding defined by the standard, UTF-16, allows to represent all Unicode code points using one or two 16-bit units.

The primitive data type char in the Java programming language is an unsigned 16-bit integer that can represent a Unicode code point in the range U+0000 to U+FFFF, or the code units of UTF-16. The various types and classes in the Java platform that represent character sequences - `char[]`, implementations of `java.lang.CharSequence` (such as the `String` class), and implementations of `java.text.CharacterIterator` - are UTF-16 sequences. Most Java source code is written in ASCII, a 7-bit character encoding, or ISO-8859-1, an 8-bit character encoding, but is translated into UTF-16 before processing.

The `Character` class is an object wrapper for the `char` primitive type. The `Character` class also contains static methods such as `isLowerCase()` and `isDigit()` for determining the properties of a character. These methods have overloads that accept either a char (which allows representation of Unicode code points in the range U+0000 to U+FFFF) or an int (which allows representation of all Unicode code points).

Locale Identification and Localization

A `Locale` object is an identifier for a particular combination of language and region. Localization is the process of adapting software for a specific region or language by adding locale-specific components and translating text.

Locales

On the Java platform, a locale is simply an identifier for a particular combination of language and region. It is not a collection of locale-specific attributes. Instead, each locale-sensitive class maintains its own locale-specific information. With this design, there is no difference in how user and system objects maintain their locale-specific resources. Both use the standard localization mechanism.

Java programs are *not* assigned a single global locale. All locale-sensitive operations may be explicitly given a locale as an argument. This greatly simplifies multilingual programs. While a global locale is not enforced, a default locale is available for programs that do not wish to manage locales explicitly. A default locale also makes it possible to affect the behavior of the entire presentation with a single choice.

Java locales act as requests for certain behavior from another object. For example, a French Canadian locale passed to a `Calendar` object asks that the `Calendar` behave correctly for the customs of Quebec. It is up to the object accepting the locale to do the right thing. If the object has not been localized for a particular locale, it will try to find a "close" match with a locale for which it has been localized. Thus if a `Calendar` object was not localized for French Canada, but was localized for the French language in general, it would use the French localization instead.

Locale Class

A `Locale` object represents a specific geographical, political, or cultural region. An operation that requires a locale to perform its task is called locale-sensitive and uses the `Locale` object to tailor information for the user. For example, displaying a number is a locale-sensitive operation - the number should be formatted according to the customs and conventions of the user's native country, region, or culture.

Supported Locales

On the Java Platform, there does not have to be a single set of supported locales, since each class maintains its own localizations. Nevertheless, there is a consistent set of localizations supported by the classes of the Java Platform. Other implementations of the Java Platform may support different locales. Locales that are supported by the JDK and JRE are summarized by release, using the search field for the Oracle Technology Network page, search for "Supported Locales".

Localized Resources

All locale-sensitive classes must be able to access resources customized for the locales they support. To aid in the process of localization, it helps to have these resources grouped together by locale and separated from the locale-neutral parts of the program.

ResourceBundle Class

The class `ResourceBundle` is an abstract base class representing containers of resources. Programmers create subclasses of `ResourceBundle` that contain resources for a particular locale. New resources can be added to an instance of `ResourceBundle`, or new instances of ResourceBundle can be added to a system without affecting the code that uses them. Packaging resources as classes allows developers to take advantage of Java's class loading mechanism to find resources.

Resource bundles contain locale-specific objects. When a program needs a locale-specific resource, such as a `String` object, the program can load it from the resource bundle that is appropriate for the current user's locale. In this way, the programmer can write code that is largely independent of the user's locale, isolating most, if not all, of the locale-specific information in resource bundles.

This allows Java programmers to write code that can:

- be easily localized, or translated, into different languages
- handle multiple locales at once
- be easily modified later to support even more locales

ResourceBundle.Control Class

`ResourceBundle.Control` is a nested class of ResourceBundle. It defines methods to be called by the `ResourceBundle.getBundle` factory methods so that the resource bundle loading behavior may be changed. For example, application specific resource bundle formats, such as XML, could be supported by overriding the methods.

Since Java SE 9, `ResourceBundle.Control` is not supported in named modules. Existing code using `Control` is expected to work, but for new code in a named module, implement *`basename`*`Provider` and load the resource bundle from there. See Resource Bundles and Named Modules.

ListResourceBundle Class

`ListResourceBundle` is an abstract subclass of `ResourceBundle` that manages resources for a locale in a convenient and easy to use list.

PropertyResourceBundle Class

`PropertyResourceBundle` is a concrete subclass of `ResourceBundle` that manages resources for a locale using a set of static strings from a property file.

Date and Time Handling

The Date-Time package, `java.time`, introduced in Java SE 8, provides a comprehensive model for date and time. Although java.time is based on the International Organization for Standardization (ISO) calendar system, commonly used global calendars are also supported.

See The Date-Time Packages lesson in The Java Tutorials (Java SE 8 and earlier).

Text Processing

Text processing involves formatting locale-sensitive information such as, currencies, dates, times, and text messages. It also includes manipulating text in a locale-sensitive manner, meaning that string operations, such as searching and sorting, are properly performed regardless of locale.

Formatting

It is in formatting data for output that many cultural conventions are applied. Numbers, dates, times, and messages may all require formatting before they can be displayed. The Java platform provides a set of flexible formatting classes that can handle both the standard locale formats and programmer defined custom formats. These formatting classes are also able to parse formatted strings back into their constituent objects.

Format Class

The class `Format` is an abstract base class for formatting locale-sensitive information such as dates, times, messages, and numbers. Three main subclasses are provided: `DateFormat`, `NumberFormat`, and `MessageFormat`. These three also provide subclasses of their own.

DateFormat Class

Dates and times are stored internally in a locale-independent way, but should be formatted so that they can be displayed in a locale-sensitive manner. For example, the same date might be formatted as:

- November 3, 1997 (English)
- 3 novembre 1997 (French)

The class `DateFormat` is an abstract base class for formatting and parsing date and time values in a locale-independent manner. It has a number of static factory methods for getting standard time formats for a given locale.

The `DateFormat` object uses `Calendar` and `TimeZone` objects in order to interpret time values. By default, a `DateFormat` object for a given locale will use the appropriate `Calendar` object for that locale and the system's default `TimeZone` object. The programmer can override these choices if desired.

SimpleDateFormat Class

The class `SimpleDateFormat` is a concrete class for formatting and parsing dates and times in a locale-sensitive manner. It allows for formatting (milliseconds to text), parsing (text to milliseconds), and normalization.

DateFormatSymbols Class

The class `DateFormatSymbols` is used to encapsulate localizable date-time formatting data, such as the names of the months, the names of the days of the week, time of day, and the time zone data. The `DateFormat` and `SimpleDateFormat` classes both use the `DateFormatSymbols` class to encapsulate this information.

Usually, programmers will not use the `DateFormatSymbols` directly. Rather, they will implement formatting with the `DateFormat` class's factory methods.

NumberFormat Class

The class `NumberFormat` is an abstract base class for formatting and parsing numeric data. It contains a number of static factory methods for getting different kinds of locale-specific number formats.

The `NumberFormat` class helps programmers to format and parse numbers for any locale. Code using this class can be completely independent of the locale conventions for decimal points, thousands-separators, the particular decimal digits used, or whether the number format is even decimal. The application can also display a number as a normal decimal number, currency, or percentage:

- 1,234.5 (decimal number in U.S. format)
- $1,234.50 (U.S. currency in U.S. format)
- 1.234,50 € (European currency in German format)
- 123.450% (percent in German format)

DecimalFormat Class

Numbers are stored internally in a locale-independent way, but should be formatted so that they can be displayed in a locale-sensitive manner. For example, when using "#,###.00" as a pattern, the same number might be formatted as:

- 1.234,56 (German)
- 1,234.56 (English)

The class `DecimalFormat`, which is a concrete subclass of the `NumberFormat` class, can format decimal numbers. Programmers generally will not instantiate this class directly but will use the factory methods provided.

The `DecimalFormat` class has the ability to take a pattern string to specify how a number should be formatted. The pattern specifies attributes such as the precision of the number, whether leading zeros should be printed, and what currency symbols are used. The pattern string can be altered if a program needs to create a custom format.

DecimalFormatSymbols Class

The class `DecimalFormatSymbols` represents the set of symbols (such as the decimal separator, the grouping separator, and so on) needed by `DecimalFormat` to format numbers. `DecimalFormat` creates for itself an instance of `DecimalFormatSymbols` from its locale data. A programmer needing to change any of these symbols can get the `DecimalFormatSymbols` object from the `DecimalFormat` object and then modify it.

ChoiceFormat Class

The class `ChoiceFormat` is a concrete subclass of the `NumberFormat` class. The `ChoiceFormat` class allows the programmer to attach a format to a range of numbers. It is generally used in a `MessageFormat` object for handling plurals.

MessageFormat Class

Programs often need to build messages from sequences of strings, numbers and other data. For example, the text of a message displaying the number of files on a disk drive will vary:

- The disk C contains 100 files.
- The disk D contains 1 file.
- The disk F contains 0 files.

If a message built from sequences of strings and numbers is hard-coded, it cannot be translated into other languages. For example, note the different positions of the parameters "3" and "G" in the following translations:

- The disk G contains 3 files. (English)
- Il y a 3 fichiers sur le disque G. (French)

The class `MessageFormat` provides a means to produce concatenated messages in language-neutral way. The MessageFormat object takes a set of objects, formats them, and then inserts the formatted strings into the pattern at the appropriate places.

ParsePosition Class

The class `ParsePosition` is used by the Format class and its subclasses to keep track of the current position during parsing. The `parseObject()` method in the `Format` class requires a `ParsePosition` object as an argument.

FieldPosition Class

The `FieldPosition` class is used by the `Format` class and its subclasses to identify fields in formatted output. One version of the `format()` method in the `Format` class requires a `FieldPosition` object as an argument.

Locale-Sensitive String Operations

Programs frequently need to manipulate strings. Common operations on strings include searching and sorting. Some tasks, such as collating strings or finding various

boundaries in text, are surprisingly difficult to get right and are even more difficult when multiple languages must be considered. The Java Platform provides classes for handling many of these common string manipulations in a locale-sensitive manner.

Collator Class

The `Collator` class performs locale-sensitive string comparison. Programmers use this class to build searching and alphabetical sorting routines for natural language text. `Collator` is an abstract base class. Its subclasses implement specific collation strategies. One subclass, `RuleBasedCollator`, is applicable to a wide set of languages. Other subclasses may be created to handle more specialized needs.

RuleBasedCollator Class

The `RuleBasedCollator` class, which is a concrete subclass of the `Collator` class, provides a simple, data-driven, table collator. Using `RuleBasedCollator`, a programmer can create a customized table-based collator. For example, a programmer can build a collator that will ignore (or notice) uppercase letters, accents, and Unicode combining characters.

CollationElementIterator Class

The `CollationElementIterator` class is used as an iterator to walk through each character of an international string. Programmers use the iterator to return the ordering priority of the positioned character. The ordering priority of a character, or key, defines how a character is collated in the given `Collator` object. The `CollationElementIterator` class is used by the `compare()` method of the `RuleBasedCollator` class.

CollationKey Class

A `CollationKey` object represents a string under the rules of a specific `Collator` object. Comparing two `CollationKey` objects returns the relative order of the strings they represent. Using `CollationKey` objects to compare strings is generally faster than using the `Collator.compare()` method. Thus, when the strings must be compared multiple times, for example when sorting a list of strings, it is more efficient to use `CollationKey` objects.

BreakIterator Class

The `BreakIterator` class indirectly implements methods for finding the position of the following types of boundaries in a string of text:

- potential line break
- sentence
- word
- character

The conventions on where to break lines, sentences, words, and characters vary from one language to another. Since the BreakIterator class is locale-sensitive, it can be used by programs that perform text operations. For example, consider a a word processing program that can highlight a character, cut a word, move the cursor to the next sentence, or word-wrap at a line ending. This word processing program would use break iterators to determine the logical boundaries in text, enabling it to perform text operations in a locale-sensitive manner.

StringCharacterIterator Class

The `StringCharacterIterator` class provides the ability to iterate over a string of Unicode characters in a bidirectional manner. This class uses a cursor to move within a range of text, and can return individual characters or their index values. The `StringCharacterIterator` class implements the character iterator functionality of the `CharacterIterator` interface.

CharacterIterator Interface

The `CharacterIterator` interface defines a protocol for bidirectional iteration over Unicode characters. Classes should implement this interface if they want to move about within a range of text and return individual Unicode characters or their index values. `CharacterIterator`is for searching is useful when performing character searches.

Normalizer Class

The `Normalizer` class provides methods to transform Unicode text into an equivalent composed or decomposed form. The class supports the *Unicode Normalization Forms* defined by the Unicode standard.

Locale-Sensitive Services SPIs

Locale sensitive services provided by classes in the java.text and java.util packages can be extended by implementing locale-sensitive services SPIs for locales the Java runtime has not yet supported.

Although JDK 9 no longer supports the extension mechanism, SPI implementations for internationalization functions in the `java.text.spi`, `java.util.spi`, and `java.awt.im.spi` packages will be loaded from the application's classpath.

In addition to localized symbols or names for the `Currency`, `Locale`, and `TimeZone` classes in the `java.util` package, implementations of the following classes in the `java.text` package can be plugged in with the SPIs.

- `BreakIterator`
- `Collator`
- `DateFormat`
- `DateFormatSymbols`
- `DecimalFormatSymbols`
- `NumberFormat`

Character Encoding Conversion

The Java platform uses Unicode as its native character encoding; however, many Java programs still need to handle text data in other encodings. Java therefore provides a set of classes that convert many standard character encodings to and from Unicode. Java programs that need to deal with non-Unicode text data convert that data into

Unicode, process the data as Unicode, then convert the result back to the external character encoding. The `InputStreamReader` and `OutputStreamWriter` classes provide methods that can convert between other character encodings and Unicode.

Supported Encodings

The `InputStreamReader`, `OutputStreamWriter`, and String classes can convert between Unicode and the set of character encodings listed in Supported Encodings.

Stream I/O

The Java Platform provides features in the `java.io` package to improve the handling of character data: the `Reader` and `Writer` classes, and an enhancement to the `PrintStream` class.

Reader and Writer Classes

The `Reader` and `Writer` class hierarchies provide the ability to perform I/O operations on character streams. These hierarchies parallel the `InputStream` and `OutputStream` class hierarchies, but operate on streams of characters rather than streams of bytes. Character streams make it easy to write programs that are not dependent upon a specific character encoding, and are therefore easier to internationalize. The `Reader` and `Writer` classes also have the ability to convert between Unicode and other character encodings.

PrintStream Class

The `PrintStream` class produces output using the system's default character encoding and line terminator. This change allows methods such as `System.out.println()` to act more reasonably with non-ASCII data.

Charset Package

The `java.nio.charset` package provides the underpinnings for character encoding conversion. Applications can use its classes to fine-tune the behavior of built-in character converters. Developers can also create custom converters for character encodings that are not supported by built-in character converters, using the `java.nio.charset.spi` package.

Input Methods

Input methods are software components that let the user enter text in ways other than simple typing on a keyboard. They are commonly used to enter Japanese, Chinese, or Korean - languages using thousands of different characters - on keyboards with far fewer keys. However, the Java platform also supports input methods for other languages and the use of entirely different input mechanisms, such as handwriting or speech recognition.

The Java platform enables the use of native input methods provided by the host operating system, such as Windows or Solaris, as well as the implementation and use of input methods written in the Java programming language.

The term input methods does not refer to class methods of the Java programming language.

Input Method Support in Swing

The Swing text components provide an integrated user interface for text input using input methods. Depending on the locale, one of two input styles is used. With on-the-spot (inline) input, the style used for most locales, the input methods insert the text directly into the text component while the text is being composed. With below-the-spot input, the style used for Chinese locales, a separate composition window is used, which is positioned automatically to be near the point where the text is to be inserted after being committed.

An application using Swing text components does not have to coordinate the interaction between the text components and input methods. However, it should call `InputContext.endComposition` when all text must be committed, such as when a document is saved or printed.

Input Method Framework

The input method framework enables the collaboration between text editing components and input methods in entering text. Programmers who develop text editing components or input methods use this framework. Other application developers generally make only minimal use of it. For example, they should call `InputContext.endComposition` when all text must be committed, such as when a document is saved or printed.

3
Supported Encodings

The `java.io.InputStreamReader`, `java.io.OutputStreamWriter`, `java.lang.String` classes, and classes in the `java.nio.charset` package can convert between Unicode and a number of other character encodings. The supported encodings vary between different implementations of the Java SE Platform. The class description for `java.nio.charset.Charset` lists the encodings that any implementation of the Java SE Platform is required to support.

The following tables show the encoding sets supported by this version of the Oracle Java SE Platform. The canonical names used by the `java.nio` APIs are in many cases not the same as those used in the `java.io` and `java.lang` APIs.

Basic Encoding Set (contained in java.base module)

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
CESU-8	CESU8	CESU8 csCESU-8	Unicode CESU-8
GB18030	GB18030	gb18030-2022 or gb18030-2000 if the system property and value `jdk.charset.GB18030=2000` are specified	Simplified Chinese, PRC standard
IBM00858	Cp858	cp858 ccsid00858 cp00858 858 PC-Multilingual-850+euro	Variant of Cp850 with Euro character
IBM437	Cp437	cp437 ibm437 ibm-437 437 cspc8codepage437 windows-437	MS-DOS United States, Australia, New Zealand, South Africa
IBM775	Cp775	cp775 ibm775 ibm-775 775	PC Baltic
IBM850	Cp850	cp850 ibm-850 ibm850 850 cspc850multilingual	MS-DOS Latin-1
IBM852	Cp852	cp852 ibm852 ibm-852 852 csPCp852	MS-DOS Latin-2
IBM855	Cp855	cp855 ibm-855 ibm855 855 cspcp855	IBM Cyrillic
IBM857	Cp857	cp857 ibm857 ibm-857 857 csIBM857	IBM Turkish
IBM862	Cp862	cp862 ibm862 ibm-862 862 csIBM862 cspc862latinhebrew	PC Hebrew
IBM866	Cp866	cp866 ibm866 ibm-866 866 csIBM866	MS-DOS Russian

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
ISO-8859-1	ISO8859_1	iso-ir-100 ISO_8859-1 latin1 l1 IBM819 cp819 csISOLatin1 819 IBM-819 ISO8859_1 ISO_8859-1:1987 ISO_8859_1 8859_1 ISO8859-1	ISO-8859-1, Latin Alphabet No. 1
ISO-8859-13	ISO8859_13	iso8859_13 8859_13 iso_8859-13 ISO8859-13	Latin Alphabet No. 7
ISO-8859-15	ISO8859_15	ISO_8859-15 Latin-9 csISO885915 8859_15 ISO-8859-15 ISO8859_15 ISO8859-15 IBM923 IBM-923 cp923 923 LATIN0 LATIN9 L9 csISOlatin0 csISOlatin9 ISO8859_15_FDIS	Latin Alphabet No. 9
ISO-8859-16	ISO8859_16	iso-ir-226 ISO_8859-16:2001 ISO_8859-16 latin10 l10 csISO885916	Latin Alphabet No. 10 or South-Eastern European
ISO-8859-2	ISO8859_2	iso8859_2 8859_2 iso-ir-101 ISO_8859-2 ISO_8859-2:1987 ISO8859-2 latin2 l2 ibm912 ibm-912 cp912 912 csISOLatin2	Latin Alphabet No. 2
ISO-8859-4	ISO8859_4	iso8859_4 iso8859-4 8859_4 iso-ir-110 ISO_8859-4 ISO_8859-4:1988 latin4 l4 ibm914 ibm-914 cp914 914 csISOLatin4	Latin Alphabet No. 4
ISO-8859-5	ISO8859_5	iso8859_5 8859_5 iso-ir-144 ISO_8859-5 ISO_8859-5:1988 ISO8859-5 cyrillic ibm915 ibm-915 cp915 915 csISOLatinCyrillic	Latin/Cyrillic Alphabet
ISO-8859-7	ISO8859_7	iso8859_7 8859_7 iso-ir-126 ISO_8859-7 ISO_8859-7:1987 ELOT_928 ECMA-118 greek greek8 csISOLatinGreek sun_eu_greek ibm813 ibm-813 813 cp813 iso8859-7	Latin/Greek Alphabet (ISO-8859-7:2003)

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
ISO-8859-9	ISO8859_9	iso8859_9 8859_9 iso-ir-148 ISO_8859-9 ISO_8859-9:1989 ISO8859-9 latin5 l5 ibm920 ibm-920 920 cp920 csISOLatin5	Latin Alphabet No. 5
KOI8-R	KOI8_R	koi8_r koi8 cskoi8r	KOI8-R, Russian
KOI8-U	KOI8_U	koi8_u	KOI8-U, Ukrainian
US-ASCII	ASCII	iso-ir-6 ANSI_X3.4-1986 ISO_646.irv:1991 ASCII ISO646-US us IBM367 cp367 csASCII default 646 iso_646.irv:1983 ANSI_X3.4-1968 ascii7	American Standard Code for Information Interchange
UTF-16	UTF-16	UTF_16 utf16 unicode UnicodeBig	Sixteen-bit Unicode (or UCS) Transformation Format, byte order identified by an optional byte-order mark
UTF-16BE	UnicodeBigUnmarked	UTF_16BE ISO-10646-UCS-2 X-UTF-16BE UnicodeBigUnmarked	Sixteen-bit Unicode (or UCS) Transformation Format, big-endian byte order
UTF-16LE	UnicodeLittleUnmarked	UTF_16LE X-UTF-16LE UnicodeLittleUnmarked	Sixteen-bit Unicode (or UCS) Transformation Format, little-endian byte order
UTF-32	UTF-32	UTF_32 UTF32	32-bit Unicode (or UCS) Transformation Format, byte order identified by an optional byte-order mark
UTF-32BE	UTF-32BE	UTF_32BE X-UTF-32BE	32-bit Unicode (or UCS) Transformation Format, big-endian byte order
UTF-32LE	UTF-32LE	UTF_32LE X-UTF-32LE	32-bit Unicode (or UCS) Transformation Format, little-endian byte order
UTF-8	UTF8	UTF8 unicode-1-1-utf-8	Eight-bit Unicode (or UCS) Transformation Format
windows-1250	Cp1250	cp1250 cp5346	Windows Eastern European
windows-1251	Cp1251	cp1251 cp5347 ansi-1251	Windows Cyrillic
windows-1252	Cp1252	cp1252 cp5348 ibm-1252 ibm1252	Windows Latin-1
windows-1253	Cp1253	cp1253 cp5349	Windows Greek
windows-1254	Cp1254	cp1254 cp5350	Windows Turkish
windows-1257	Cp1257	cp1257 cp5353	Windows Baltic

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
x-IBM737	Cp737	cp737 ibm737 ibm-737 737	PC Greek
x-IBM874	Cp874	cp874 ibm874 ibm-874 874	IBM Thai
x-UTF-16LE-BOM	UnicodeLittle	UnicodeLittle	Sixteen-bit Unicode (or UCS) Transformation Format, little-endian byte order, with byte-order mark
X-UTF-32BE-BOM	X-UTF-32BE-BOM	UTF_32BE_BOM UTF-32BE-BOM	32-bit Unicode (or UCS) Transformation Format, big-endian byte order, with byte-order mark
X-UTF-32LE-BOM	X-UTF-32LE-BOM	UTF_32LE_BOM UTF-32LE-BOM	32-bit Unicode (or UCS) Transformation Format, little-endian byte order, with byte-order mark

Extended Encoding Set (contained in jdk.charsets module)

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
Big5	Big5	csBig5	Big5, Traditional Chinese
Big5-HKSCS	Big5_HKSCS	Big5_HKSCS big5hk big5-hkscs big5hkscs	Big5 with Hong Kong extensions, Traditional Chinese (incorporating 2001 revision)
EUC-JP	EUC_JP	euc_jp eucjis eucjp Extended_UNIX_Code_Packed_Format_for_Japanese csEUCPkdFmtjapanese x-euc-jp x-eucjp	JISX 0201, 0208 and 0212, EUC encoding Japanese
EUC-KR	EUC_KR	euc_kr ksc5601 euckr ks_c_5601-1987 ksc5601-1987 ksc5601_1987 ksc_5601 csEUCKR 5601	KS C 5601, EUC encoding, Korean
GB2312	EUC_CN	gb2312 gb2312-80 gb2312-1980 euc-cn euccn x-EUC-CN EUC_CN	GB2312, EUC encoding, Simplified Chinese
GBK	GBK	windows-936 CP936	GBK, Simplified Chinese
IBM01140	Cp1140	cp1140 ccsid01140 cp01140 1140 ebcdic-us-037+euro	Variant of Cp037 with Euro character

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
IBM01141	Cp1141	cp1141 ccsid01141 cp01141 1141 ebcdic-de-273+euro	Variant of Cp273 with Euro character
IBM01142	Cp1142	cp1142 ccsid01142 cp01142 1142 ebcdic-no-277+euro ebcdic-dk-277+euro	Variant of Cp277 with Euro character
IBM01143	Cp1143	cp1143 ccsid01143 cp01143 1143 ebcdic-fi-278+euro ebcdic-se-278+euro	Variant of Cp278 with Euro character
IBM01144	Cp1144	cp1144 ccsid01144 cp01144 1144 ebcdic-it-280+euro	Variant of Cp280 with Euro character
IBM01145	Cp1145	cp1145 ccsid01145 cp01145 1145 ebcdic-es-284+euro	Variant of Cp284 with Euro character
IBM01146	Cp1146	cp1146 ccsid01146 cp01146 1146 ebcdic-gb-285+euro	Variant of Cp285 with Euro character
IBM01147	Cp1147	cp1147 ccsid01147 cp01147 1147 ebcdic-fr-277+euro	Variant of Cp297 with Euro character
IBM01148	Cp1148	cp1148 ccsid01148 cp01148 1148 ebcdic-international-500+euro	Variant of Cp500 with Euro character
IBM01149	Cp1149	cp1149 ccsid01149 cp01149 1149 ebcdic-s-871+euro	Variant of Cp871 with Euro character
IBM037	Cp037	cp037 ibm037 ebcdic-cp-us ebcdic-cp-ca ebcdic-cp-wt ebcdic-cp-nl csIBM037 cs-ebcdic-cp-us cs-ebcdic-cp-ca cs-ebcdic-cp-wt cs-ebcdic-cp-nl ibm-037 ibm-37 cpibm37 037	USA, Canada (Bilingual, French), Netherlands, Portugal, Brazil, Australia
IBM1026	Cp1026	cp1026 ibm1026 ibm-1026 1026	IBM Latin-5, Turkey
IBM1047	Cp1047	cp1047 ibm-1047 1047	Latin-1 character set for EBCDIC hosts
IBM273	Cp273	cp273 ibm273 ibm-273 273	IBM Austria, Germany
IBM277	Cp277	cp277 ibm277 ibm-277 277	IBM Denmark, Norway
IBM278	Cp278	cp278 ibm278 ibm-278 278 ebcdic-sv ebcdic-cp-se csIBM278	IBM Finland, Sweden
IBM280	Cp280	cp280 ibm280 ibm-280 280	IBM Italy

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
IBM284	Cp284	cp284 ibm284 ibm-284 284 csIBM284 cpibm284	IBM Catalan/Spain, Spanish Latin America
IBM285	Cp285	cp285 ibm285 ibm-285 285 ebcdic-cp-gb ebcdic-gb csIBM285 cpibm285	IBM United Kingdom, Ireland
IBM290	Cp290	cp290 ibm290 ibm-290 csIBM290 EBCDIC-JP-kana 290	IBM Japanese Katakana Host Extended SBCS
IBM297	Cp297	cp297 ibm297 ibm-297 297 ebcdic-cp-fr cpibm297 csIBM297	IBM France
IBM420	Cp420	cp420 ibm420 ibm-420 ebcdic-cp-ar1 420 csIBM420	IBM Arabic
IBM424	Cp424	cp424 ibm424 ibm-424 424 ebcdic-cp-he csIBM424	IBM Hebrew
IBM500	Cp500	cp500 ibm500 ibm-500 500 ebcdic-cp-ch ebcdic-cp-bh csIBM500	EBCDIC 500V1
IBM860	Cp860	cp860 ibm860 ibm-860 860 csIBM860	MS-DOS Portuguese
IBM861	Cp861	cp861 ibm861 ibm-861 861 csIBM861 cp-is	MS-DOS Icelandic
IBM863	Cp863	cp863 ibm863 ibm-863 863 csIBM863	MS-DOS Canadian French
IBM864	Cp864	cp864 ibm864 ibm-864 864 csIBM864	PC Arabic
IBM865	Cp865	cp865 ibm865 ibm-865 865 csIBM865	MS-DOS Nordic
IBM868	Cp868	cp868 ibm868 ibm-868 868 cp-ar csIBM868	MS-DOS Pakistan
IBM869	Cp869	cp869 ibm869 ibm-869 869 cp-gr csIBM869	IBM Modern Greek
IBM870	Cp870	cp870 ibm870 ibm-870 870 ebcdic-cp-roece ebcdic-cp-yu csIBM870	IBM Multilingual Latin-2
IBM871	Cp871	cp871 ibm871 ibm-871 871 ebcdic-cp-is csIBM871	IBM Iceland

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
IBM918	Cp918	cp918 ibm-918 918 ebcdic-cp-ar2	IBM Pakistan (Urdu)
IBM-Thai	Cp838	cp838 ibm838 ibm-838 838	IBM Thailand extended SBCS
ISO-2022-CN	ISO2022CN	ISO2022CN csISO2022CN	GB2312 and CNS11643 in ISO 2022 CN form, Simplified and Traditional Chinese (conversion to Unicode only)
ISO-2022-JP	ISO2022JP	iso2022jp jis csISO2022JP jis_encoding csjisencoding	JIS X 0201, 0208, in ISO 2022 form, Japanese
ISO-2022-JP-2	ISO2022JP2	csISO2022JP2 iso2022jp2	JIS X 0201, 0208, 0212 in ISO 2022 form, Japanese
ISO-2022-KR	ISO2022KR	ISO2022KR csISO2022KR	ISO 2022 KR, Korean
ISO-8859-3	ISO8859_3	iso8859_3 8859_3 ISO_8859-3:1988 iso-ir-109 ISO_8859-3 ISO8859-3 latin3 l3 ibm913 ibm-913 cp913 913 csISOLatin3	Latin Alphabet No. 3
ISO-8859-6	ISO8859_6	iso8859_6 8859_6 iso-ir-127 ISO_8859-6 ISO_8859-6:1987 ISO8859-6 ECMA-114 ASMO-708 arabic ibm1089 ibm-1089 cp1089 1089 csISOLatinArabic	Latin/Arabic Alphabet
ISO-8859-8	ISO8859_8	iso8859_8 8859_8 iso-ir-138 ISO_8859-8 ISO_8859-8:1988 ISO8859-8 cp916 916 ibm916 ibm-916 hebrew csISOLatinHebrew	Latin/Hebrew Alphabet
JIS_X0201	JIS_X0201	JIS0201 JIS_X0201 X0201 csHalfWidthKatakana	JIS X 0201
JIS_X0212-1990	JIS0212	JIS0212 jis_x0212-1990 x0212 iso-ir-159 csISO159JISX02121990	JIS X 0212
Shift_JIS	SJIS	sjis shift_jis shift-jis ms_kanji x-sjis csShiftJIS	Shift-JIS, Japanese

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
TIS-620	TIS620	tis620 tis620.2533	TIS620, Thai
windows-1255	Cp1255	cp1255	Windows Hebrew
windows-1256	Cp1256	cp1256	Windows Arabic
windows-1258	Cp1258	cp1258	Windows Vietnamese
windows-31j	MS932	MS932 windows-932 csWindows31J	Windows Japanese
x-Big5-HKSCS-2001	x-Big5-HKSCS-2001	Big5_HKSCS_2001 big5hk-2001 big5-hkscs-2001 big5-hkscs:unicode3.0 big5hkscs-2001	Big5 with Hong Kong Supplementary Character Set, 2001 revision
x-Big5-Solaris	Big5_Solaris	Big5_Solaris	Big5 with seven additional Hanzi ideograph character mappings for the Solaris zh_TW.BIG5 locale
x-euc-jp-linux	EUC_JP_LINUX	euc_jp_linux euc-jp-linux	JISX 0201, 0208, EUC encoding Japanese
x-eucJP-Open	EUC_JP_Solaris	EUC_JP_Solaris eucJP-open	JISX 0201, 0208, 0212, EUC encoding Japanese
x-EUC-TW	EUC_TW	euc_tw euctw cns11643 EUC-TW	CNS11643 (Plane 1-7,15), EUC encoding, Traditional Chinese
x-IBM1006	Cp1006	cp1006 ibm1006 ibm-1006 1006	IBM AIX Pakistan (Urdu)
x-IBM1025	Cp1025	cp1025 ibm1025 ibm-1025 1025	IBM Multilingual Cyrillic: Bulgaria, Bosnia, Herzegovinia, Macedonia (FYR)
x-IBM1046	Cp1046	cp1046 ibm1046 ibm-1046 1046	IBM Arabic - Windows
x-IBM1097	Cp1097	cp1097 ibm1097 ibm-1097 1097	IBM Iran (Farsi)/ Persian
x-IBM1098	Cp1098	cp1098 ibm1098 ibm-1098 1098	IBM Iran (Farsi)/ Persian (PC)
x-IBM1112	Cp1112	cp1112 ibm1112 ibm-1112 1112	IBM Latvia, Lithuania
x-IBM1122	Cp1122	cp1122 ibm1122 ibm-1122 1122	IBM Estonia
x-IBM1123	Cp1123	cp1123 ibm1123 ibm-1123 1123	IBM Ukraine
x-IBM1124	Cp1124	cp1124 ibm1124 ibm-1124 1124	IBM AIX Ukraine
x-IBM1129	Cp1129	cp1129 ibm1129 ibm-1129 1129	IBM AIX Vietnamese

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
x-IBM1166	Cp1166	cp1166 ibm1166 ibm-1166 1166	IBM Cyrillic Multilingual with euro for Kazakhstan
x-IBM1364	Cp1364	cp1364 ibm1364 ibm-1364 1364	IBM EBCDIC KS X 1005-1
x-IBM1381	Cp1381	cp1381 ibm1381 ibm-1381 1381	IBM OS/2, DOS People's Republic of China (PRC)
x-IBM1383	Cp1383	cp1383 ibm1383 ibm-1383 1383 ibmeuccn ibm-euccn cpeuccn	IBM AIX People's Republic of China (PRC)
x-IBM300	Cp300	cp300 ibm300 ibm-300 300	IBM Japanese Latin Host Double-Byte
x-IBM33722	Cp33722	cp33722 ibm33722 ibm-33722 ibm-5050 ibm-33722_vascii_vpua 33722	IBM-eucJP - Japanese (superset of 5050)
x-IBM833	Cp833	cp833 ibm833 ibm-833	IBM Korean Host Extended SBCS
x-IBM834	Cp834	cp834 ibm834 834 ibm-834	IBM EBCDIC DBCS-only Korean
x-IBM856	Cp856	cp856 ibm-856 ibm856 856	IBM Hebrew
x-IBM875	Cp875	cp875 ibm875 ibm-875 875	IBM Greek
x-IBM921	Cp921	cp921 ibm921 ibm-921 921	IBM Latvia, Lithuania (AIX, DOS)
x-IBM922	Cp922	cp922 ibm922 ibm-922 922	IBM Estonia (AIX, DOS)
x-IBM930	Cp930	cp930 ibm930 ibm-930 930	Japanese Katakana-Kanji mixed with 4370 UDC, superset of 5026
x-IBM933	Cp933	cp933 ibm933 ibm-933 933	Korean Mixed with 1880 UDC, superset of 5029
x-IBM935	Cp935	cp935 ibm935 ibm-935 935	Simplified Chinese Host mixed with 1880 UDC, superset of 5031
x-IBM937	Cp937	cp937 ibm937 ibm-937 937	Traditional Chinese Host miexed with 6204 UDC, superset of 5033
x-IBM939	Cp939	cp939 ibm939 ibm-939 939	Japanese Latin Kanji mixed with 4370 UDC, superset of 5035
x-IBM942	Cp942	cp942 ibm942 ibm-942 942	IBM OS/2 Japanese, superset of Cp932

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
x-IBM942C	Cp942C	cp942C ibm942C ibm-942C 942C cp932 ibm932 ibm-932 932 x-ibm932	Variant of Cp942
x-IBM943	Cp943	cp943 ibm943 ibm-943 943	IBM OS/2 Japanese, superset of Cp932 and Shift-JIS
x-IBM943C	Cp943C	cp943C ibm943C ibm-943C 943C	Variant of Cp943
x-IBM948	Cp948	cp948 ibm948 ibm-948 948	OS/2 Chinese (Taiwan) superset of 938
x-IBM949	Cp949	cp949 ibm949 ibm-949 949	PC Korean
x-IBM949C	Cp949C	cp949C ibm949C ibm-949C 949C	Variant of Cp949
x-IBM950	Cp950	cp950 ibm950 ibm-950 950	PC Chinese (Hong Kong, Taiwan)
x-IBM964	Cp964	cp964 ibm964 ibm-964 ibm-euctw 964	AIX Chinese (Taiwan)
x-IBM970	Cp970	cp970 ibm970 ibm-970 ibm-eucKR 970	AIX Korean
x-ISCII91	ISCII91	iscii ST_SEV_358-88 iso-ir-153 csISO153GOST1976874 ISCII91	ISCII91 encoding of Indic scripts
x-ISO-2022-CN-CNS	ISO2022CN_CNS	ISO2022CN_CNS ISO-2022-CN-CNS	CNS11643 in ISO 2022 CN form, Traditional Chinese (conversion from Unicode only)
x-ISO-2022-CN-GB	ISO2022CN_GB	ISO2022CN_GB ISO-2022-CN-GB	GB2312 in ISO 2022 CN form, Simplified Chinese (conversion from Unicode only)
x-iso-8859-11	x-iso-8859-11	iso-8859-11 iso8859_11	Latin/Thai Alphabet
x-JIS0208	JIS0208	JIS0208 JIS_C6226-1983 iso-ir-87 x0208 JIS_X0208-1983 csISO87JISX0208	JIS X 0208
x-JISAutoDetect	JISAutoDetect	JISAutoDetect	Detects and converts from Shift-JIS, EUC-JP, ISO 2022 JP (conversion to Unicode only)
x-Johab	x-Johab	ksc5601-1992 ksc5601_1992 ms1361 johab	Korean, Johab character set

Canonical Name for java.nio API	Canonical Name for java.io API and java.lang API	Alias or Aliases	Description
x-MacArabic	MacArabic	MacArabic	Macintosh Arabic
x-MacCentralEurope	MacCentralEurope	MacCentralEurope	Macintosh Latin-2
x-MacCroatian	MacCroatian	MacCroatian	Macintosh Croatian
x-MacCyrillic	MacCyrillic	MacCyrillic	Macintosh Cyrillic
x-MacDingbat	MacDingbat	MacDingbat	Macintosh Dingbat
x-MacGreek	MacGreek	MacGreek	Macintosh Greek
x-MacHebrew	MacHebrew	MacHebrew	Macintosh Hebrew
x-MacIceland	MacIceland	MacIceland	Macintosh Iceland
x-MacRoman	MacRoman	MacRoman	Macintosh Roman
x-MacRomania	MacRomania	MacRomania	Macintosh Romania
x-MacSymbol	MacSymbol	MacSymbol	Macintosh Symbol
x-MacThai	MacThai	MacThai	Macintosh Thai
x-MacTurkish	MacTurkish	MacTurkish	Macintosh Turkish
x-MacUkraine	MacUkraine	MacUkraine	Macintosh Ukraine
x-MS932_0213	x-MS950-HKSCS	MS932-0213 MS932_0213 MS932:2004 windows-932-0213 windows-932:2004	Shift_JISX0213 Windows MS932 Variant
x-MS950-HKSCS	MS950_HKSCS	MS950_HKSCS	Windows Traditional Chinese with Hong Kong extensions
x-MS950-HKSCS-XP	x-mswin-936	MS950_HKSCS_XP	HKSCS Windows XP Variant
x-mswin-936	MS936	ms936 ms_936	Windows Simplified Chinese
x-PCK	PCK	pck	Solaris version of Shift_JIS
x-SJIS_0213	x-SJIS_0213	sjis-0213 sjis_0213 sjis:2004 sjis_0213:2004 shift_jis_0213:2004 shift_jis:2004	Shift_JISX0213
x-windows-50220	MS50220	ms50220 cp50220	Windows Codepage 50220 (7-bit implementation)
x-windows-50221	MS50221	ms50221 cp50221	Windows Codepage 50221 (7-bit implementation)
x-windows-874	MS874	ms874 ms-874 windows-874	Windows Thai
x-windows-949	MS949	ms949 windows949 windows-949 ms_949	Windows Korean
x-windows-950	MS950	ms950 windows-950	Windows Traditional Chinese
x-windows-iso2022jp	windows-iso2022jp	windows-iso2022jp	Variant ISO-2022-JP (MS932 based)

Printing Charset Information

The following applications print the aliases and the canonical name for `java.io` and `java.lang` APIs of each charset supported by Java SE.

The following application prints the aliases of each charset:

```
import java.nio.charset.*;

class DisplayCharsetAliases {
    public static void main(String[] args) {
        System.out.println("Charset -> Aliases");
        System.out.println("==================");
        for (Charset cs : Charset.availableCharsets().values()) {
            System.out.println(cs.name() + " -> " + cs.aliases());
        }
    }
}
```

The following application prints the canonoical name for `java.io` and `java.lang` APIs of each charset:

```
import java.nio.charset.*;
import sun.nio.cs.*;

class PrintCanonicalName {
    public static void main(String[] args) {
        for (Charset cs : Charset.availableCharsets().values()) {
            System.out.println(cs.name() + ":" +
                (cs instanceof HistoricallyNamedCharset ?
                    ((HistoricallyNamedCharset)cs).historicalName() :
"-----"));
        }
    }
}
```

Compile this application as follows:

```
javac --add-exports java.base/sun.nio.cs=ALL-UNNAMED
PrintCanonicalName.java
```

4
Supported Calendars

The core of the Date-Time API is the java.time package. The classes defined in java.time base their calendar system on the ISO calendar, which is the world standard for representing date and time. The ISO calendar follows the proleptic Gregorian rules. There are also non-ISO calendars predefined in `java.time.chrono` package: the Japanese, Hijrah, Minguo, and Thai Buddhist calendars. For more about the Date-Time API, see the Internationalization Trail in the Java Tutorials.

5

Supported Fonts

Different OS platforms may provide fonts that are implemented using different font technologies. To support cross-platform use, the Java SE API defines five families of "logical" fonts that can safely be used by an application using any Java SE implementation. The names of these families are defined in the `Font` class description.

Additionally a Java SE implementation may expose the platform fonts to be used directly by name. These fonts are called "physical" fonts.

For more information on the terminology used here, see the `Font` class description.

Support for Physical Fonts

The JDK supports TrueType, OpenType, and PostScript Type 1 fonts.

Physical fonts need to be installed in locations known to the Java runtime environment. The JDK locates fonts in the standard font locations defined by the host operating system.

You can add physical fonts that use a supported font technology by installing them in a way supported by the host operating system. The recommended location to add per-user fonts on Solaris or Linux is the `$HOME/.fonts` directory which is searched by the platform's `libfontconfig`, and which is in turn used by the JDK.

Support for Logical Fonts

Typically one logical font maps to several physical fonts in order to cover a larger range of code points than is possible with a single font. Logical fonts are mapped to physical fonts in implementation-dependent ways, and can vary from platform to platform and from release to release.

Font configuration files are used by some implementations to handle the mapping, see Font Configuration Files:

- Current releases for Windows always use font configuration files.
- The macOS implementation always ignores font configuration files.
- Releases for Solaris and Linux use font configuration files only if there is an exact match for the OS version, otherwise font configuration files are ignored and platform APIs are used to populate the logical fonts.

6
Font Configuration Files

The Java Platform defines five logical font names that every implementation must support: `Serif`, `SansSerif`, `Monospaced`, `Dialog`, and `DialogInput`. These logical font names are mapped to physical fonts in implementation dependent ways.

One way the Oracle JDK maps logical font names to physical fonts is by using *font configuration* files. There may be several files to support different mappings depending on the host operating system version. The files are distributed with the JDK installation. You can edit or create your own font configuration files to adjust the mappings to your particular system setup, however these must be placed in `conf/fonts`, and are subject to implementation notes discussed below.

Font configuration files come in two formats: a properties format and a binary format. The properties format is described in detail in this document and can be used for user-defined configurations. The binary format is undocumented and used only for the JDK's predefined configurations; the corresponding files in properties format are available for reference as files with the `.properties.src` extension.

Supported Platforms

Font configuration files are implementation dependent. Not all implementations of the Java Platform use them, and the format and content vary between different runtime environments as well as between releases. The macOS implementation does not use font configuration files, as the mapping is hard coded in the source and cannot be changed in any way.

The Oracle JDK supports font configuration files on the host operating system as follows:

- For Windows, font configuration files are required.
- For macOS, font configuration files are unsupported.
- For Linux and Solaris: the Oracle JDK is moving away from providing custom font configuration files on Linux platforms, as they are difficult to keep up to date across distributions and versions. A distribution that has control over the fonts on the system can continue to provide this custom file. If the JRE finds a custom file that exactly matches the distribution and version it will use it. If no exact match is found, the JRE dynamically creates the file at runtime. These generated files are placed in a location determined by the implementation. They should be considered implementation internal: they are not user editable and do not follow the syntax as described in this specification.

Loading Font Configuration Files

The JDK places any files that it provides in `$JDKHOME/lib`. Do not modify that location. Instead, put any updates or custom versions of the configuration files in `$JDKHOME/conf/fonts`. If you provide a custom configuration file, it must adhere to the implementation limitation that a font cannot contain more than 254 slots.

On platforms that support font configuration files, the runtime will look first in `$JDKHOME/conf/fonts`. In other words, a user-supplied file is preferred if it is a match.

The font configuration file for a host operating system is located as follows:

- *JavaHome* - the JDK directory, as given by the `java.home` system property.
- *OS* - a string identifying an operating system variant:
  - For Windows, empty.
  - For Solaris, empty.
  - For Linux, `"RedHat"`, `"SuSE"`, etc.
- *Version* - a string identifying the operating system version.

The runtime uses the first of the following files it finds:

```
JavaHome/lib/fontconfig.OS.Version.properties
JavaHome/lib/fontconfig.OS.Version.bfc
JavaHome/lib/fontconfig.OS.properties
JavaHome/lib/fontconfig.OS.bfc
JavaHome/lib/fontconfig.Version.properties
JavaHome/lib/fontconfig.Version.bfc
JavaHome/lib/fontconfig.properties
JavaHome/lib/fontconfig.bfc
```

Files with a `.properties` suffix are assumed to be properties files as specified by the `Properties` class and are loaded through that class. Files without this suffix are assumed to be in binary format.

Names Used in Font Configuration Files

Throughout the font configuration files, a number of different names are used:

- *LogicalFontName* - one of the five logical font names: `serif`, `sansserif`, `monospaced`, `dialog`, and `dialoginput`. In font configuration files, these names are always in lowercase.
- *StyleName* - one of the four standard font styles: `plain`, `bold`, `italic`, and `bolditalic`. Again, these names are always in lowercase.
- *PlatformFontName* - the name of a physical font, in a format typically used on the platform:
  - On Windows, a font face name, such as `"Courier New"` or `"\uad74\ub9bc"`.
  - On Solaris and Linux, an xlfd name, such as `"-monotype-times new roman-regular-r---*-%d-*-*-p-*-iso8859-1"`. Note that `"%d"` is used for the font size - the actual font size is filled in at runtime.
- *CharacterSubsetName* - a name for a subset of the Unicode character set which certain component fonts can render. For Windows, the following names are predefined: alphabetic, arabic, chinese-ms936, chinese-gb18030, chinese-ms950, chinese-hkscs, cyrillic-iso8859-5, cyrillic-cp1251, cyrillic-koi8-r, devanagari, dingbats, greek, hebrew, japanese, korean, latin, symbol, thai. For Solaris and Linux, the following names are predefined: arabic, chinese-gb2312, chinese-gbk, chinese-gb18030-0, chinese-gb18030-1, chinese-cns11643-1, chinese-cns11643-2, chinese-cns11643-3, chinese-big5, chinese-hkscs, cyrillic, devanagari, dingbats, greek, hebrew, japanese-x0201, japanese-x0208, japanese-x0212, korean, korean-johab, latin-1, latin-2, latin-4, latin-5, latin-7, latin-9,

symbol,thai. A font configuration file may define additional names to identify additional character subsets.

- *Encoding* - the canonical name of the default encoding, as provided by `java.nio.charset.Charset.defaultCharset().name()`.
- *Language* - the language of the initial default locale.
- *Country* - the country of the initial default locale.

Properties for All Platforms

Properties that are applicable to all platforms enable you to specify the font configuration format version, component font mappings, search sequences, exclusion ranges, proportional fonts, font file names, and appended font path.

Version Property

The version property identifies the font configuration format version. This document specifies version 1.

The complete property has the form:

```
version=1
```

Component Font Mappings

Component font mapping properties describe which physical font to use to render characters from a given character subset with a given logical font in a given style.

The keys have the forms:

```
allfonts.CharacterSubsetName
LogicalFontName.StyleName.CharacterSubsetName
```

The first form is used if the same font is used for a character subset independent of logical font and style (in this case, the font rendering engines apply algorithmic styles to the font). The second form is used if different physical fonts are used for a character subset for different logical fonts and styles. In this case, properties must be specified for each combination of logical font and style, so 20 properties for one character subset. If a property of the first form is present for a character subset, then properties of the second form for the same character subset are ignored.

The values are platform font names, as described in Names Used in Font Configuration Files.

Since the character subsets supported by given fonts often overlap, separate search sequence properties are used to define in which order to try the fonts when rendering a character.

Search Sequences

The Java runtime uses sequence properties to determine search sequences for the five logical fonts. However, a font configuration file may specify properties that are specific to a

combination of encoding, language, and country, and the runtime will then use a lookup to determine the search sequence property for each logical font.

The keys have the form:

```
sequence.allfonts.Encoding.Language.Country
sequence.LogicalFontName.Encoding.Language.Country
sequence.allfonts.Encoding.Language
sequence.LogicalFontName.Encoding.Language
sequence.allfonts.Encoding
sequence.LogicalFontName.Encoding
sequence.allfonts
sequence.LogicalFontName
```

The `allfonts` forms are used if the sequence is used for all five logical fonts. The forms specifying logical font names are used if different sequences are used for different logical fonts.

For each logical font, the Java runtime uses the property value with the first of the above keys. This property determines the primary search sequence for the logical font.

The file may also define a single fallback search sequence. The key for the fallback search sequence property is:

```
sequence.fallback
```

The values of all search sequence properties have the form:

```
SearchSequenceValue:
  CharacterSubsetName
  CharacterSubsetName , SearchSequenceValue
```

The primary search sequence properties specify character subset names for required fonts, which are used for both AWT and 2D font rendering. The fallback search sequence property gives character subset names for optional fonts, which are used as fallbacks for all logical fonts, but only for 2D font rendering. On Windows, if there is a system EUDC (End User Defined Characters) font registered with Windows, the runtime automatically adds this font as well as a fallback font for 2D rendering.

The sequence properties determine in which sequence component fonts are tried to render a given character. For example, given the following properties:

```
sequence.monospaced=japanese,alphabetic
sequence.fallback=korean
monospaced.plain.alphabetic=Arial
monospaced.plain.japanese=MSGothic
monospaced.plain.korean=Gulim
```

The runtime will first attempt to render a character with the MSGothic font. If that font doesn't provide a glyph for the character, it will attempt the Arial font. For 2D rendering, it will also try the Gulim font as well as any TrueType, OpenType, or Type 1 fonts in the system's standard font locations. For 2D rendering on Windows, if there is a system EUDC font registered with Windows, the runtime will also try this EUDC font.

When calculating font metrics for a logical font without reference to a string, only the required fonts are taken into consideration. For the example above, the `FontMetrics.getMaxDescent` method would return results based on the MSGothic and Arial fonts, but not the Gulim font. In this way, simple user interface elements such as buttons, which sometimes calculate their size based on font metrics, are not affected by an extended list of component fonts which their labels usually don't use. On the other hand, text components typically calculate metrics based on the text they contain and thus will obtain correct results.

The sequence properties that the runtime obtains for the five logical fonts should list the same character subsets, but may list them in different order.

Exclusion Ranges

The exclusion range properties specify Unicode character ranges which should be excluded from being rendered with the fonts corresponding to a given character subset. This is used if a font with a large character repertoire needs to be placed early in the search sequence (for example, for performance reasons), but some characters that it supports should be drawn with a different font instead. These properties are optional, so there's at most one per character subset.

The keys have the form:

```
exclusion.CharacterSubsetName
```

The values have the form:

```
ExclusionRangeValue:
  Range
  Range , ExclusionRangeValue

Range:
  Char - Char

Char:
  HexDigit HexDigit HexDigit HexDigit
  HexDigit HexDigit HexDigit HexDigit HexDigit HexDigit
```

A *Char* is a Unicode character represented as a hexadecimal value.

Proportional Fonts

The proportional font properties describe the relationship between proportional and non-proportional variants of otherwise equivalent fonts. These properties are used to implement preferences specified by the `GraphicsEnvironment.preferProportionalFonts` method.

The keys have the form:

```
proportional.PlatformFontName
```

Space characters in the platform font name must be replaced with underscore characters (_).

The values have the form:

```
PlatformFontName
```

In values, space characters are left unmodified.

Each property indicates that the font named in the value is the proportional equivalent of the font named in the key, and also that the font named in the key is the non-proportional equivalent of the font named in the value.

Font File Names

Font file name properties provide the names of the files containing the physical fonts used in the font configuration file. File names are required for all physical fonts on Windows and recommended for all physical fonts on Solaris and Linux.

The keys have the form:

```
filename.PlatformFontName
```

Space characters in the platform font name must be replaced with underscore characters (_).

The values are the file names of the files containing the fonts. On Windows, simple file names are used; and the runtime environment looks for each file first in its own `lib/fonts` directory, then in the Windows fonts directory. On Solaris and Linux, absolute path names, path names starting with `"$JRE_LIB_FONTS"` for the runtime environment's own `lib/fonts` directory, or xlfd names are used.

Appended Font Path

The Java runtime can automatically determine a number of directories that contain font files, such as its own `lib/fonts` directory or the Windows fonts folder. Additional directories can be specified to be appended to the font path.

The key has the form:

```
appendedfontpath
```

The value has the form:

```
AppendedFontPathValue:
  Directory
  Directory PathSeparator AppendedFontPathValue
```

The path separator is the platform dependent value of `java.io.File.pathSeparator`.

Properties for Windows

There are no platform-specific properties for Windows. However, there is a special form of the character subset name used in search sequences. The name "alphabetic" can take a suffix indicating the character encoding associated with the subset:

```
alphabetic
alphabetic/default
alphabetic/1252
```

This information is only used for AWT, not for 2D. The `/default` suffix restricts use of the component fonts for this character subset to the character set of the default encoding; the `/1252` suffix to the Windows-1252 character set. For accessing component font mappings and exclusion ranges, the character encoding suffix is omitted. For all other character subsets, the AWT character encoding is determined internally by the Java runtime.

Property for Solaris and Linux

The only property that is specific to Solaris and Linux is the AWT font path, which identifies platform directories that should be added to the X11 server font path.

The keys have the form:

```
awtfontpath.CharacterSubsetName
```

The values have the form:

```
AWTFontPathValue:
  Directory
  Directory : AWTFontPathValue
```

The directories must be valid X11 font directories. The Java runtime ensures that the directories for all character subsets of a primary search sequence found by the search sequence lookup (see Search Sequences) are part of the X11 font path. The implementation assumes that all logical fonts use the same set of character subsets for a given environment of encoding, language, and country.