OpenJDK
Installing
Contributing
Sponsoring
Developers' Guide
Vulnerabilities
JDK GA/EA Builds
Mailing lists
Wiki · IRC
Bylaws · Census
Legal
Workshop
JEP Process
Source code
Mercurial
GitHub
Tools
Git
jtreg harness
Groups
(overview)
Adoption
Build
Client Libraries
Compatibility & Specification Review
Compiler
Conformance
Core Libraries
Governing Board
HotSpot
IDE Tooling & Support
Internationalization
JMX
Members
Networking
Porters
Quality
Security
Serviceability
Vulnerability
Web
Projects
(overview, archive)
Amber
Babylon
CRaC
Caciocavallo
Closures
Code Tools
Coin
Common VM Interface
Compiler Grammar
Detroit
Developers' Guide
Device I/O
Duke
Galahad
Graal
IcedTea
JDK 7
JDK 8
JDK 8 Updates
JDK 9
JDK (…, 21, 22, 23)
JDK Updates
JavaDoc.Next
Jigsaw
Kona
Kulla
Lambda
Lanai
Leyden
Lilliput
Locale Enhancement
Loom
Memory Model Update
Metropolis
Mission Control
Multi-Language VM
Nashorn
New I/O
OpenJFX
Panama
Penrose
Port: AArch32
Port: AArch64
Port: BSD
Port: Haiku
Port: Mac OS X
Port: MIPS
Port: Mobile
Port: PowerPC/AIX
Port: RISC-V
Port: s390x
Portola
SCTP
Shenandoah
Skara
Sumatra
Tiered Attribution
Tsan
Type Annotations
Valhalla
Verona
VisualVM
Wakefield
Zero
ZGC
ORACLE

JEP 400: UTF-8 by Default

Authors	Alan Bateman, Naoto Sato
Owner	Naoto Sato
Type	Feature
Scope	SE
Status	Closed / Delivered
Release	18
Component	core-libs / java.nio.charsets
Discussion	core dash libs dash dev at openjdk dot java dot net
Effort	XS
Duration	XS
Reviewed by	Alex Buckley, Brian Goetz
Endorsed by	Brian Goetz
Created	2017/08/31 13:16
Updated	2023/06/12 13:46
Issue	8187041

Summary

Specify UTF-8 as the default charset of the standard Java APIs. With this change, APIs that depend upon the default charset will behave consistently across all implementations, operating systems, locales, and configurations.

Goals

- Make Java programs more predictable and portable when their code relies on the default charset.
- Clarify where the standard Java API uses the default charset.
- Standardize on UTF-8 throughout the standard Java APIs, except for console I/O.

Non-Goals

- It is not a goal to define new standard Java APIs or supported JDK APIs, although this effort may identify opportunities where new convenience methods might make existing APIs more approachable or easier to use.
- There is no intent to deprecate or remove standard Java APIs that rely on the default charset rather than taking an explicit charset parameter.

Motivation

Standard Java APIs for reading and writing files and for processing text allow a *charset* to be passed as an argument. A charset governs the conversion between raw bytes and the 16-bit `char` values of the Java programming language. Supported charsets include, for example, US-ASCII, UTF-8, and ISO-8859-1.

If a charset argument is not passed, then standard Java APIs typically use the *default charset*. The JDK chooses the default charset at startup based upon the run-time environment: the operating system, the user's locale, and other factors.

Because the default charset is not the same everywhere, APIs that use the default charset pose many non-obvious hazards, even to experienced developers.

Consider an application that creates a `java.io.FileWriter` without passing a charset, and then uses it to write some text to a file. The resulting file will contain a sequence of bytes encoded using the default charset of the JDK running the application. A second application, run on a different machine or by a different user on the same machine, creates a `java.io.FileReader` without passing a charset and uses it to read the bytes in that file. The resulting text contains a sequence of characters decoded using the default charset of the JDK running the second application. If the default charset differs between the JDK of the first application and the JDK of the second application then the resulting text may be silently corrupted or incomplete, since the `FileReader` cannot tell that it decoded the text using the *wrong* charset relative to the `FileWriter`. Here is an example of this hazard, where a Japanese text file encoded in `UTF-8` on macOS is corrupted when read on Windows in US-English or Japanese locales:

```
java.io.FileReader(“hello.txt”) -> “こんにちは” (macOS)
java.io.FileReader(“hello.txt”) -> “ã?“ã,“ã?«ã?¡ã? ” (Windows (en-US))
java.io.FileReader(“hello.txt”) -> “縺ォ縺。縺ッ” (Windows (ja-JP)
```

Developers familiar with such hazards can use methods and constructors that take a charset argument explicitly. However, having to pass an argument prevents methods and constructors from being used via method references (::) in stream pipelines.

Developers sometimes attempt to configure the default charset by setting the system property `file.encoding` on the command line (i.e., `java -Dfile.encoding=...`), but this has never been supported. Furthermore, attempting to set the property programmatically (i.e., `System.setProperty(...)`) after the Java runtime has started does not work.

Not all standard Java APIs defer to the JDK's choice of default charset. For example, the methods in `java.nio.file.Files` that read or write files without a `Charset` argument are specified to always use UTF-8. The fact that newer APIs default to using UTF-8 while older APIs default to using the default charset is a hazard for applications that use a mix of APIs.

The entire Java ecosystem would benefit if the default charset were specified to be the same everywhere. Applications that are not concerned with portability will see little impact, while applications that embrace portability by passing charset arguments will see no impact. UTF-8 has long been the most common charset on the World Wide Web. UTF-8 is standard for the XML and JSON files processed by vast numbers of Java programs, and Java's own APIs increasingly favor UTF-8 in, e.g., the NIO API and for property files. It therefore makes sense to specify UTF-8 as the default charset for all Java APIs.

We recognize that this change could have a widespread compatibility impact on programs that migrate to JDK 18. For this reason, it will always be possible to recover the pre-JDK 18 behavior, where the default charset is environment-dependent.

Description

In JDK 17 and earlier, the default charset is determined when the Java runtime starts. On macOS, it is UTF-8 except in the POSIX C locale. On other operating systems, it depends upon the user's locale and the default encoding, e.g., on Windows, it is a codepage-based charset such as `windows-1252` or `windows-31j`. The method `java.nio.charsets.Charset.defaultCharset()` returns the default charset. A quick way to see the default charset of the current JDK is with the following command:

```
java -XshowSettings:properties -version 2>&1 | grep file.encoding
```

Several standard Java APIs use the default charset, including:

- In the `java.io` package, `InputStreamReader`, `FileReader`, `OutputStreamWriter`, `FileWriter`, and `PrintStream` define constructors to create readers, writers, and print streams that encode or decode using the default charset.
- In the `java.util` package, `Formatter` and `Scanner` define constructors whose results use the default charset.
- In the `java.net` package, `URLEncoder` and `URLDecoder` define deprecated methods that use the default charset.

We propose to change the specification of `Charset.defaultCharset()` to say that the default charset is UTF-8 unless configured otherwise by an implementation-specific means. (See below for how to configure the JDK.) The UTF-8 charset is specified by RFC 2279; the transformation format upon which it is based is specified in Amendment 2 of ISO 10646-1 and is also described in the Unicode Standard. It is not to be confused with Modified UTF-8.

We will update the specifications of all standard Java APIs that use the default charset to cross-reference `Charset.defaultCharset()`. Those APIs include the ones listed above, but not `System.out` and `System.err`, whose charset will be as specified by `Console.charset()`.

The `file.encoding` and `native.encoding` system properties

As envisaged by the specification of `Charset.defaultCharset()`, the JDK will allow the default charset to be configured to something other than UTF-8. We will revise the treatment of the system property `file.encoding` so that setting it on the command line is the supported means of configuring the default charset. We will specify this in an implementation note of `System.getProperties()` as follows:

- If `file.encoding` is set to `"COMPAT"` (i.e., `java -Dfile.encoding=COMPAT`), then the default charset will be the charset chosen by the algorithm in JDK 17 and earlier, based on the user's operating system, locale, and other factors. The value of `file.encoding` will be set to the name of that charset.
- If `file.encoding` is set to `"UTF-8"` (i.e., `java -Dfile.encoding=UTF-8`), then the default charset will be UTF-8. This no-op value is defined in order to preserve the behavior of existing command lines.
- The treatment of values other than `"COMPAT"` and `"UTF-8"` are not specified. They are not supported, but if such a value worked in JDK 17 then it will likely continue to work in JDK 18.

Prior to deploying on a JDK where UTF-8 is the default charset, developers are strongly encouraged to check for charset issues by starting the Java runtime with `java -Dfile.encoding=UTF-8 ...` on their current JDK (8-17).

JDK 17 introduced the `native.encoding` system property as a standard way for programs to obtain the charset chosen by the JDK's algorithm, regardless of whether the default charset is actually configured to be that charset. In JDK 18, if `file.encoding` is set to `COMPAT` on the command line, then the run-time value of `file.encoding` will be the same as the run-time value of `native.encoding`; if `file.encoding` is set to `UTF-8` on the command line, then the run-time value of `file.encoding` may differ from the run-time value of `native.encoding`.

In *Risks and Assumptions* below, we discuss how to mitigate the possible incompatibilities that arise from this change to `file.encoding`, as well as the `native.encoding` system property and recommendations for applications.

There are three charset-related system properties used internally by the JDK. They remain unspecified and unsupported, but are documented here for completeness:

- `sun.stdout.encoding` and `sun.stderr.encoding` — the names of the charsets used for the standard output stream (`System.out`) and standard error stream (`System.err`), and in the `java.io.Console` API.
- `sun.jnu.encoding` — the name of the charset used by the implementation of `java.nio.file` when encoding or decoding filename paths, as opposed to file contents. On macOS its value is `"UTF-8"`; on other platforms it is typically the default charset.

Source file encoding

The Java language allows source code to express Unicode characters in a UTF-16 encoding, and this is unaffected by the choice of UTF-8 for the default charset. However, the `javac` compiler is affected because it assumes that `.java` source files are encoded with the default charset, unless configured otherwise by the `-encoding` option. If source files were saved with a non-UTF-8 encoding and compiled with an earlier JDK, then recompiling on JDK 18 or later may cause problems. For example, if a non-UTF-8 source file has string literals that contain non-ASCII characters, then those literals may be misinterpreted by `javac` in JDK 18 or later unless `-encoding` is used.

Prior to compiling on a JDK where UTF-8 is the default charset, developers are strongly encouraged to check for charset issues by compiling with `javac -encoding UTF-8 ...` on their current JDK (8-17). Alternatively, developers who prefer to save source files with a non-UTF-8 encoding can prevent `javac` from assuming UTF-8 by setting the `-encoding` option to the value of the `native.encoding` system property on JDK 17 and later.

The legacy `default` charset

In JDK 17 and earlier, the name `default` is recognized as an alias for the `US-ASCII` charset. That is, `Charset.forName("default")` produces the same result as `Charset.forName("US-ASCII")`. The `default` alias was introduced in JDK 1.5 to ensure that legacy code which used `sun.io` converters could migrate to the `java.nio.charset` framework introduced in JDK 1.4.

It would be extremely confusing for JDK 18 to preserve `default` as an alias for `US-ASCII` when the default charset is specified to be `UTF-8`. It would also be confusing for `default` to mean `US-ASCII` when the user configures the default charset to its pre-JDK 18 value by setting `-Dfile.encoding=COMPAT` on the command line. Redefining `default` to be an alias not for `US-ASCII` but rather for the default charset (whether `UTF-8` or user-configured) would cause subtle behavioral changes in the (few) programs that call `Charset.forName("default")`.

We believe that continuing to recognize `default` in JDK 18 would be prolonging a poor decision. It is not defined by the Java SE Platform, nor is it recognized by IANA as the name or alias of any character set. In fact, for ASCII-based network protocols, IANA encourages use of the canonical name `US-ASCII` rather than just `ASCII` or obscure aliases such as `ANSI_X3.4-1968` -- plainly, use of the JDK-specific alias `default` goes counter to that advice. Java programs can use the enum constant `StandardCharsets.US_ASCII` to make their intent clear, rather than passing a string to `Charset.forName(...)`.

Accordingly, in JDK 18, `Charset.forName("default")` will throw an `UnsupportedCharsetException`. This will give developers a chance to detect use of the idiom and migrate to either `US-ASCII` or to the result of `Charset.defaultCharset()`.

Testing

- Significant testing is required to understand the extent of the compatibility impact of this change. Testing by developers or organizations with geographically diverse user populations will be needed.
- Developers can check for issues with an existing JDK release by running with `-Dfile.encoding=UTF-8` in advance of any early-access or GA release with this change.

Risks and Assumptions

We assume that applications in many environments will see no impact from Java's choice of `UTF-8`:

- On macOS, the default charset has been UTF-8 for several releases, except when configured to use the POSIX C locale.
- In many Linux distributions, though not all, the default charset is UTF-8, so no change will be discernible in those environments.
- Many server applications are already started with `-Dfile.encoding=UTF-8`, so they will not experience any change.

In other environments, the risk of changing the default charset to `UTF-8` after more than 20 years may be significant. The most obvious risk is that applications which implicitly depend on the default charset (e.g., by not passing an explicit charset argument to APIs) will behave incorrectly when processing data produced when the default charset was unspecified. A further risk is that data corruption may silently occur. We expect the main impact will be to users of Windows in Asian locales, and possibly some server environments in Asian and other locales. Possible scenarios include:

- If an application that has been running for years with `windows-31j` as the default charset is upgraded to a JDK release that uses UTF-8 as the default charset then it will experience problems when reading files that are encoded in `windows-31j`. In this case, the application code could be changed to pass the `windows-31j` charset when opening such files. If the code cannot be changed, then starting the Java runtime with `-Dfile.encoding=COMPAT` will force the default charset to be `windows-31j` until the application is updated or the files are converted to UTF-8.
- In environments where several JDK versions are in use, users might not be able to exchange file data. If, e.g., one user uses an older JDK release where `windows-31j` is the default and another uses a newer JDK where UTF-8 is the default, then text files created by the first user might not be readable by the second. In this case the user on the older JDK release could specify `-Dfile.encoding=UTF-8` when starting applications, or the user on the newer release could specify `-Dfile.encoding=COMPAT`.

Where application code can be changed, then we recommend it is changed to pass a charset argument to constructors. If an application has no particular preference among charsets, and is satisfied with the traditional environment-driven selection for the default charset, then the following code can be used *on all Java releases* to obtain the charset determined from the environment:

```
String encoding = System.getProperty("native.encoding");  // Populated on Java 18 and later
Charset cs = (encoding != null) ? Charset.forName(encoding) : Charset.defaultCharset();
var reader = new FileReader("file.txt", cs);
```

If neither application code nor Java startup can be changed, then it will be necessary to inspect the application code to determine manually whether it will run compatibly on JDK 18.

Alternatives

- *Preserve the status quo* — This does not eliminate the hazards described above.
- *Deprecate all methods in the Java API that use the default charset* — This would encourage developers to use constructors and methods that take a charset parameter, but the resulting code would be more verbose.
- *Specify UTF-8 as the default charset without providing any means to change it* — The compatibility impact of this change would be too high.

© 2024 Oracle Corporation and/or its affiliates
Terms of Use · License: GPLv2 · Privacy · Trademarks