ECMA-402, 6th edition, June 2019

ECMAScript® 2019 Internationalization API Specification

Contributing to this Specification

This specification is developed on GitHub with the help of the ECMAScript community. There are a number of ways to contribute to the development of this specification:

GitHub Repository: https://github.com/tc39/ecma402
Issues: All Issues, File a New Issue
Pull Requests: All Pull Requests, Create a New Pull Request
Test Suite: Test262
Editor:

- Leo Balter (@leobalter)

Community:

- Mailing list: es-discuss
- IRC: #tc39 on freenode
- IRC: #tc39-ecma402 on freenode

Refer to the colophon for more information on how this document is created.

Introduction

This specification's source can be found at https://github.com/tc39/ecma402.

The ECMAScript 2019 Internationalization API Specification (ECMA-402 6th Edition), provides key language sensitive functionality as a complement to the ECMAScript 2020 Language Specification (ECMA-262 10th Edition or successor). Its functionality has been selected from that of well-established internationalization APIs such as those of the Internationalization Components for Unicode (ICU) library, of the .NET framework, or of the Java platform.

The 1st Edition API was developed by an ad-hoc group established by Ecma TC39 in September 2010 based on a proposal by Nebojša Ćirić and Jungshik Shin.

The 2nd Edition API was adopted by the General Assembly of June 2015, as a complement to the ECMAScript 6th Edition.

The 3rd Edition API was the first edition released under Ecma TC39's new yearly release cadence and open development process. A plain-text source document was built from the ECMA-402 source document to serve as the base for further development entirely on GitHub. Over the year of this standard's development, dozens of pull requests and issues were filed representing several of bug fixes, editorial fixes and other improvements. Additionally, numerous software tools were developed to aid in this effort including Ecmarkup, Ecmarkdown, and Grammarkdown.

Dozens of individuals representing many organizations have made very significant contributions within Ecma TC39 to the development of this edition and to the prior editions. In addition, a vibrant community has emerged supporting TC39's ECMAScript efforts. This community has reviewed numerous drafts, filed dozens of bug reports, performed implementation experiments, contributed test suites, and educated the world-wide developer community about ECMAScript Internationalization. Unfortunately, it is impossible to identify and acknowledge every person and organization who has contributed to this effort.

Norbert Lindenberg
ECMA-402, 1st Edition Project Editor

Rick Waldron
ECMA-402, 2nd Edition Project Editor

Caridy Patiño
ECMA-402, 3rd, 4th and 5th Editions Project Editor

Caridy Patiño, Daniel Ehrenberg, Leo Balter
ECMA-402, 6th Edition Project Editors

1 Scope

This Standard defines the application programming interface for ECMAScript objects that support programs that need to adapt to the linguistic and cultural conventions used by different human languages and countries.

2 Conformance

A conforming implementation of the ECMAScript 2020 Internationalization API Specification must conform to the ECMAScript 2020 Language Specification (ECMA-262 10th Edition, or successor), and must provide and support all the objects, properties, functions, and program semantics described in this specification.

A conforming implementation of the ECMAScript 2020 Internationalization API Specification is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of the ECMAScript 2020 Internationalization API Specification is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification. A conforming implementation is not permitted to add optional arguments to the functions defined in this specification.

A conforming implementation is permitted to accept additional values, and then have implementation-defined behaviour instead of throwing a **RangeError**, for the following properties of *options* arguments:

- The *options* property localeMatcher in all constructors and supportedLocalesOf methods.
- The *options* properties usage and sensitivity in the Collator constructor.
- The *options* properties style and currencyDisplay in the NumberFormat constructor.
- The *options* properties minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, and maximumSignificantDigits in the NumberFormat constructor, provided that the additional values are interpreted as integer values higher than the specified limits.
- The *options* properties listed in Table 5 in the DateTimeFormat constructor.
- The *options* property formatMatcher in the DateTimeFormat constructor.
- The *options* property type in the PluralRules constructor.

3 Normative References

The following referenced documents are required for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ECMAScript 2020 Language Specification (ECMA-262 10th Edition, or successor).
https://www.ecma-international.org/publications/standards/Ecma-262.htm

> NOTE
> Throughout this document, the phrase “ES2020, *x*” (where x is a sequence of numbers separated by periods) may be used as shorthand for "ECMAScript 2020 Language Specification (ECMA-262 10th Edition, sub clause *x*)".

- ISO/IEC 10646:2014: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2015 and Amendment 2, plus additional amendments and corrigenda, or successor
  - https://www.iso.org/iso/catalogue_detail.htm?csnumber=63182
  - https://www.iso.org/iso/catalogue_detail.htm?csnumber=65047
  - https://www.iso.org/iso/catalogue_detail.htm?csnumber=66791
- ISO 4217:2015, Codes for the representation of currencies and funds, or successor
- IETF BCP 47:
  - RFC 5646, Tags for Identifying Languages, or successor
  - RFC 4647, Matching of Language Tags, or successor
- IETF RFC 6067, BCP 47 Extension U, or successor
- IANA Time Zone Database
- The Unicode Standard
- Unicode Technical Standard 35, Unicode Locale Data Markup Language

4 Overview

This section contains a non-normative overview of the ECMAScript 2020 Internationalization API Specification.

4.1 Internationalization, Localization, and Globalization

Internationalization of software means designing it such that it supports or can be easily adapted to support the needs of users speaking different languages and having different cultural expectations, and enables worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization of software is commonly understood to be the combination of internationalization and localization. Globalization starts at the lowest level by using a text representation that supports all languages in the world, and using standard identifiers to identify languages, countries, time zones, and other relevant parameters. It continues with using a user interface language and data presentation that the user understands, and finally often requires product-specific adaptations to the user's language, culture, and environment.

The ECMAScript 2020 Language Specification lays the foundation by using Unicode for text representation and by providing a few language-sensitive functions, but gives applications little control over the behaviour of these functions. The ECMAScript 2020 Internationalization API Specification builds on this by providing a set of customizable language-sensitive functionality. The API is useful even for applications that themselves are not internationalized, as even applications targeting only one language and one region need to properly support that one language and region. However, the API also enables applications that support multiple languages and regions, even concurrently, as may be needed in server environments.

4.2 API Overview

The ECMAScript 2020 Internationalization API Specification is designed to complement the ECMAScript 2020 Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript 2020 Language Specification (ECMA-262 10th Edition, or successor).

The ECMAScript 2020 Internationalization API Specification provides several key pieces of language-sensitive functionality that are required in most applications: String comparison (collation), number formatting, date and time formatting, pluralization rules, and case conversion. While the ECMAScript 2020 Language Specification provides functions for this basic functionality (on Array.prototype: toLocaleString; on String.prototype: localeCompare, toLocaleLowerCase, toLocaleUpperCase; on Number.prototype: toLocaleString; on Date.prototype: toLocaleString, toLocaleDateString, and toLocaleTimeString), it leaves the actual behaviour of these functions largely up to implementations to define. The ECMAScript 2020 Internationalization API Specification provides additional functionality, control over the language and over details of the behaviour to be used, and a more complete specification of required functionality.

Applications can use the API in two ways:

1. Directly, by using the constructors Intl.Collator, Intl.NumberFormat, Intl.DateTimeFormat, or Intl.PluralRules to construct an object, specifying a list of preferred languages and options to configure the behaviour of the resulting object. The object then provides a main function (compare, select, or format), which can be called repeatedly. It also provides a resolvedOptions function, which the application can use to find out the exact configuration of the object.
2. Indirectly, by using the functions of the ECMAScript 2020 Language Specification mentioned above. The collation and formatting functions are respecified in this specification to accept the same arguments as the Collator, NumberFormat, and DateTimeFormat constructors and produce the same results as their compare or format methods. The case conversion functions are respecified to accept a list of preferred languages.

The Intl object is used to package all functionality defined in the ECMAScript 2020 Internationalization API Specification to

avoid name collisions.

4.3 Implementation Dependencies

Due to the nature of internationalization, the API specification has to leave several details implementation dependent:

- *The set of locales that an implementation supports with adequate localizations:* Linguists estimate the number of human languages to around 6000, and the more widely spoken ones have variations based on regions or other parameters. Even large locale data collections, such as the Common Locale Data Repository, cover only a subset of this large set. Implementations targeting resource-constrained devices may have to further reduce the subset.
- *The exact form of localizations such as format patterns:* In many cases locale-dependent conventions are not standardized, so different forms may exist side by side, or they vary over time. Different internationalization libraries may have implemented different forms, without any of them being actually wrong. In order to allow this API to be implemented on top of existing libraries, such variations have to be permitted.
- *Subsets of Unicode:* Some operations, such as collation, operate on strings that can include characters from the entire Unicode character set. However, both the Unicode standard and the ECMAScript standard allow implementations to limit their functionality to subsets of the Unicode character set. In addition, locale conventions typically don't specify the desired behaviour for the entire Unicode character set, but only for those characters that are relevant for the locale. While the Unicode Collation Algorithm combines a default collation order for the entire Unicode character set with the ability to tailor for local conventions, subsets and tailorings still result in differences in behaviour.

4.3.1 Compatibility across implementations

ECMA 402 describes the schema of the data used by its functions. The data contained inside is implementation-dependent, and expected to change over time and vary between implementations. The variation is visible by programmers, and it is possible to construct programs which will depend on a particular output. However, this specification attempts to describe reasonable constraints which will allow well-written programs to function across implementations. Implementations are encouraged to continue their efforts to harmonize linguistic data.

5 Notational Conventions

This standard uses a subset of the notational conventions of the ECMAScript 2020 Language Specification (ECMA-262 10th Edition), as ES2020:

- Object Internal Methods and Internal Slots, as described in ES2020, 6.1.7.2.
- Algorithm conventions, including the use of abstract operations, as described in ES2020, 7.1, 7.2, 7.3.
- Internal Slots, as described in ES2020, 9.1.
- The List and Record Specification Type, as described in ES2020, 6.2.1.

> NOTE
> As described in the ECMAScript Language Specification, algorithms are used to precisely specify the required semantics of ECMAScript constructs, but are not intended to imply the use of any specific implementation technique. Internal slots are used to define the semantics of object values, but are not part of the API. They are defined purely for expository purposes. An

implementation of the API must behave as if it produced and operated upon internal slots in the manner described here.

As an extension to the Record Specification Type, the notation "[[<*name*>]]" denotes a field whose name is given by the variable *name*, which must have a String value. For example, if a variable *s* has the value **"a"**, then [[<*s*>]] denotes the field [[a]].

This specification uses blocks demarcated as Normative Optional to denote the sense of Annex B in ECMA 262. That is, normative optional sections are required when the ECMAScript host is a web browser. The content of the section is normative but optional if the ECMAScript host is not a web browser.

5.1 Well-Known Intrinsic Objects

The following table extends the Well-Known Intrinsic Objects table defined in ES2020, 6.1.7.4.

Table 1: Well-known Intrinsic Objects (Extensions)

Intrinsic Name	Global Name	ECMAScript Language Association
%Date_now%	**`Date.now`**	The initial value of the **`now`** data property of the intrinsic %Date% (ES2020, 20.3.3.1)
%Intl%	**`Intl`**	The **`Intl`** object (8).
%Collator%	**`Intl.Collator`**	The **`Intl.Collator`** constructor (10.1)
%CollatorPrototype%	**`Intl.Collator.prototype`**	The initial value of the **`prototype`** data property of the intrinsic %Collator% (10.2.1).
%NumberFormat%	**`Intl.NumberFormat`**	The **`Intl.NumberFormat`** constructor (11.2)
%NumberFormatPrototype%	**`Intl.NumberFormat.prototype`**	The initial value of the **`prototype`** data property of the intrinsic %NumberFormat% (11.3.1).
%DateTimeFormat%	**`Intl.DateTimeFormat`**	The **`Intl.DateTimeFormat`** constructor (12.2).
%DateTimeFormatPrototype%	**`Intl.DateTimeFormat.prototype`**	The initial value of the **`prototype`** data property of the intrinsic %DateTimeFormat% (12.3.1).
%PluralRules%	**`Intl.PluralRules`**	The **`Intl.PluralRules`** constructor (13.2).
%PluralRulesPrototype%	**`Intl.PluralRules.prototype`**	The initial value of the **`prototype`** data property of the intrinsic %PluralRules% (13.3.1).
%StringProto_indexOf%	**`String.prototype.indexOf`**	The initial value of the **`indexOf`** data property of the intrinsic %StringPrototype% (ES2020, 21.1.3.8)

6 Identification of Locales, Currencies, and Time Zones

This clause describes the String values used in the ECMAScript 2020 Internationalization API Specification to identify locales, currencies, and time zones.

6.1 Case Sensitivity and Case Mapping

The String values used to identify locales, currencies, and time zones are interpreted in a case-insensitive manner, treating the Unicode Basic Latin characters **"A"** to **"Z"** (U+0041 to U+005A) as equivalent to the corresponding Basic Latin characters **"a"** to **"z"** (U+0061 to U+007A). No other case folding equivalences are applied. When mapping to upper case, a mapping shall be used that maps characters in the range **"a"** to **"z"** (U+0061 to U+007A) to the corresponding characters in the range **"A"** to **"Z"** (U+0041 to U+005A) and maps no other characters to the latter range.

EXAMPLES **"ß"** (U+00DF) must not match or be mapped to **"SS"** (U+0053, U+0053). **"ı"** (U+0131) must not match or be mapped to **"I"** (U+0049).

6.2 Language Tags

The ECMAScript 2020 Internationalization API Specification identifies locales using language tags as by the Unicode BCP 47 locale identifiers, which may include extensions such as those registered through RFC 6067. Their canonical form is that of a Unicode BCP 47 Locale Identifier, as specified in Unicode Technical Standard #35 LDML § 3.3 BCP 47 Conformance.

Unicode BCP 47 Locale Identifiers are structurally valid when they match those syntactical formatting criteria of Unicode Technical Standard 35, section 3.2, or successor, but it is not required to validate them according to the Unicode validation data. All structurally valid language tags are valid for use with the APIs defined by this standard. However, the set of locales and thus language tags that an implementation supports with adequate localizations is implementation dependent. The constructors Collator, NumberFormat, DateTimeFormat, and PluralRules map the language tags used in requests to locales supported by their respective implementations.

6.2.1 Unicode Locale Extension Sequences

This standard uses the term **"Unicode locale extension sequence"** - as described in **unicode_locale_extensions** in Unicode BCP 47 - for any substring of a language tag that is not part of a private use subtag sequence, starts with a separator **"-"** and the singleton **"u"**, and includes the maximum sequence of following non-singleton subtags and their preceding **"-"** separators.

6.2.2 IsStructurallyValidLanguageTag (*locale*)

The IsStructurallyValidLanguageTag abstract operation verifies that the *locale* argument (which must be a String value)

- represents a well-formed Unicode BCP 47 Locale Identifier" as specified in Unicode Technical Standard 35 section 3.2, or successor,
- does not include duplicate variant subtags, and
- does not include duplicate singleton subtags.

The abstract operation returns true if *locale* can be generated from the EBNF grammar in section 3.2 of the Unicode Technical

Standard 35, or successor, starting with unicode_locale_id, and does not contain duplicate variant or singleton subtags (other than as a private use subtag). It returns false otherwise. Terminal value characters in the grammar are interpreted as the Unicode equivalents of the ASCII octet values given.

6.2.3 CanonicalizeLanguageTag (*locale*)

The CanonicalizeLanguageTag abstract operation returns the canonical and case-regularized form of the *locale* argument (which must be a String value that is a structurally valid Unicode BCP 47 Locale Identifier as verified by the IsStructurallyValidLanguageTag abstract operation). A conforming implementation shall take the steps specified in the "BCP 47 Language Tag to Unicode BCP 47 Locale Identifier" algorithm, from Unicode Technical Standard #35 LDML § 3.3.1 BCP 47 Language Tag Conversion.

6.2.4 DefaultLocale ()

The DefaultLocale abstract operation returns a String value representing the structurally valid (6.2.2) and canonicalized (6.2.3) BCP 47 language tag for the host environment's current locale.

6.3 Currency Codes

The ECMAScript 2020 Internationalization API Specification identifies currencies using 3-letter currency codes as defined by ISO 4217. Their canonical form is upper case.

All well-formed 3-letter ISO 4217 currency codes are allowed. However, the set of combinations of currency code and language tag for which localized currency symbols are available is implementation dependent. Where a localized currency symbol is not available, the ISO 4217 currency code is used for formatting.

6.3.1 IsWellFormedCurrencyCode (*currency*)

The IsWellFormedCurrencyCode abstract operation verifies that the *currency* argument (which must be a String value) represents a well-formed 3-letter ISO currency code. The following steps are taken:

1. Let *normalized* be the result of mapping *currency* to upper case as described in 6.1.
2. If the number of elements in *normalized* is not 3, return **false**.
3. If *normalized* contains any character that is not in the range `"A"` to `"Z"` (U+0041 to U+005A), return **false**.
4. Return **true**.

6.4 Time Zone Names

The ECMAScript 2020 Internationalization API Specification identifies time zones using the Zone and Link names of the IANA Time Zone Database. Their canonical form is the corresponding Zone name in the casing used in the IANA Time Zone Database.

All registered Zone and Link names are allowed. Implementations must recognize all such names, and use best available

current and historical information about their offsets from UTC and their daylight saving time rules in calculations. However, the set of combinations of time zone name and language tag for which localized time zone names are available is implementation dependent.

6.4.1 IsValidTimeZoneName (*timeZone*)

The IsValidTimeZoneName abstract operation verifies that the *timeZone* argument (which must be a String value) represents a valid Zone or Link name of the IANA Time Zone Database.

The abstract operation returns true if *timeZone*, converted to upper case as described in 6.1, is equal to one of the Zone or Link names of the IANA Time Zone Database, converted to upper case as described in 6.1. It returns false otherwise.

6.4.2 CanonicalizeTimeZoneName

The CanonicalizeTimeZoneName abstract operation returns the canonical and case-regularized form of the *timeZone* argument (which must be a String value that is a valid time zone name as verified by the IsValidTimeZoneName abstract operation). The following steps are taken:

1. Let *ianaTimeZone* be the Zone or Link name of the IANA Time Zone Database such that *timeZone*, converted to upper case as described in 6.1, is equal to *ianaTimeZone*, converted to upper case as described in 6.1.
2. If *ianaTimeZone* is a Link name, let *ianaTimeZone* be the corresponding Zone name as specified in the **`"backward"`** file of the IANA Time Zone Database.
3. If *ianaTimeZone* is **`"Etc/UTC"`** or **`"Etc/GMT"`**, return **`"UTC"`**.
4. Return *ianaTimeZone*.

The Intl.DateTimeFormat constructor allows this time zone name; if the time zone is not specified, the host environment's current time zone is used. Implementations shall support UTC and the host environment's current time zone (if different from UTC) in formatting.

6.4.3 DefaultTimeZone ()

The DefaultTimeZone abstract operation returns a String value representing the valid (6.4.1) and canonicalized (6.4.2) time zone name for the host environment's current time zone.

7 Requirements for Standard Built-in ECMAScript Objects

Unless specified otherwise in this document, the objects, functions, and constructors described in this standard are subject to the generic requirements and restrictions specified for standard built-in ECMAScript objects in the ECMAScript 2020 Language Specification, 10th edition, clause 17, or successor.

8 The Intl Object

The Intl object is the *%Intl%* intrinsic object and the initial value of the **`Intl`** property of the global object. The Intl object is a single ordinary object.

The value of the [[Prototype]] internal slot of the Intl object is the intrinsic object %ObjectPrototype%.

The Intl object is not a function object. It does not have a [[Construct]] internal method; it is not possible to use the Intl object as a constructor with the **new** operator. The Intl object does not have a [[Call]] internal method; it is not possible to invoke the Intl object as a function.

The Intl object has an internal slot, [[FallbackSymbol]], which is a new %Symbol% in the current realm with the [[Description]] **`"IntlLegacyConstructedSymbol"`**

8.1 Constructor Properties of the Intl Object

8.1.1 Intl.Collator (...)

See 10.

8.1.2 Intl.NumberFormat (...)

See 11.

8.1.3 Intl.DateTimeFormat (...)

See 12.

8.1.4 Intl.PluralRules (...)

See 13.

NOTE

In ECMA 402 v1, Intl constructors supported a mode of operation where calling them with an existing object as a receiver would transform the receiver into the relevant Intl instance with all internal slots. In ECMA 402 v2, this capability was removed, to avoid adding internal slots on existing objects. In ECMA 402 v3, the capability was re-added as "normative optional" in a mode which chains the underlying Intl instance on any object, when the constructor is called. See Issue 57 for details.

8.2 Function Properties of the Intl Object

8.2.1 Intl.getCanonicalLocales (*locales*)

When the **`getCanonicalLocales`** method is called with argument *locales*, the following steps are taken:

1. Let *ll* be ? CanonicalizeLocaleList(*locales*).
2. Return CreateArrayFromList(*ll*).

9 Locale and Parameter Negotiation

The constructors for the objects providing locale sensitive services, Collator, NumberFormat, DateTimeFormat, and PluralRules, use a common pattern to negotiate the requests represented by the locales and options arguments against the actual capabilities of their implementations. The common behaviour is described here in terms of internal slots describing the capabilities and of abstract operations using these internal slots.

9.1 Internal slots of Service Constructors

The constructors Intl.Collator, Intl.NumberFormat, Intl.DateTimeFormat, and Intl.PluralRules have the following internal slots:

- [[AvailableLocales]] is a List that contains structurally valid (6.2.2) and canonicalized (6.2.3) BCP 47 language tags identifying the locales for which the implementation provides the functionality of the constructed objects. Language tags on the list must not have a Unicode locale extension sequence. The list must include the value returned by the DefaultLocale abstract operation (6.2.4), and must not include duplicates. Implementations must include in [[AvailableLocales]] locales that can serve as fallbacks in the algorithm used to resolve locales (see 9.2.6). For example, implementations that provide a **"de-DE"** locale must include a **"de"** locale that can serve as a fallback for requests such as **"de-AT"** and **"de-CH"**. For locales that in current usage would include a script subtag (such as Chinese locales), old-style language tags without script subtags must be included such that, for example, requests for **"zh-TW"** and **"zh-HK"** lead to output in traditional Chinese rather than the default simplified Chinese. The ordering of the locales within [[AvailableLocales]] is irrelevant.
- [[RelevantExtensionKeys]] is a List of keys of the language tag extensions defined in Unicode Technical Standard 35 that are relevant for the functionality of the constructed objects.
- [[SortLocaleData]] and [[SearchLocaleData]] (for Intl.Collator) and [[LocaleData]] (for Intl.NumberFormat, Intl.DateTimeFormat, and Intl.PluralRules) are records that have fields for each locale contained in [[AvailableLocales]]. The value of each of these fields must be a record that has fields for each key contained in [[RelevantExtensionKeys]]. The value of each of these fields must be a non-empty list of those values defined in Unicode Technical Standard 35 for the given key that are supported by the implementation for the given locale, with the first element providing the default value.

EXAMPLE An implementation of DateTimeFormat might include the language tag **"th"** in its [[AvailableLocales]] internal slot, and must (according to 12.3.3) include the key **"ca"** in its [[RelevantExtensionKeys]] internal slot. For Thai, the **"buddhist"** calendar is usually the default, but an implementation might also support the calendars **"gregory"**, **"chinese"**, and **"islamicc"** for the locale **"th"**. The [[LocaleData]] internal slot would therefore at least include {[[th]]: {[[ca]]: « **"buddhist"**, **"gregory"**, **"chinese"**, **"islamicc"** »}}.

9.2 Abstract Operations

Where the following abstract operations take an *availableLocales* argument, it must be an [[AvailableLocales]] List as specified in 9.1.

9.2.1 CanonicalizeLocaleList (*locales*)

The abstract operation CanonicalizeLocaleList takes the following steps:

1. If *locales* is **undefined**, then
    a. Return a new empty List.
2. Let *seen* be a new empty List.
3. If Type(*locales*) is String, then
    a. Let *O* be CreateArrayFromList(« *locales* »).
4. Else,
    a. Let *O* be ? ToObject(*locales*).
5. Let *len* be ? ToLength(? Get(*O*, **"length"**)).
6. Let *k* be 0.
7. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be ? HasProperty(*O*, *Pk*).
    c. If *kPresent* is **true**, then
        i. Let *kValue* be ? Get(*O*, *Pk*).
        ii. If Type(*kValue*) is not String or Object, throw a **TypeError** exception.
        iii. Let *tag* be ? ToString(*kValue*).
        iv. If IsStructurallyValidLanguageTag(*tag*) is **false**, throw a **RangeError** exception.
        v. Let *canonicalizedTag* be CanonicalizeLanguageTag(*tag*).
        vi. If *canonicalizedTag* is not an element of *seen*, append *canonicalizedTag* as the last element of *seen*.
    d. Increase *k* by 1.
8. Return *seen*.

NOTE 1
Non-normative summary: The abstract operation interprets the *locales* argument as an array and copies its elements into a List, validating the elements as structurally valid language tags and canonicalizing them, and omitting duplicates.

NOTE 2
Requiring *kValue* to be a String or Object means that the Number value **NaN** will not be interpreted as the language tag **"nan"**, which stands for Min Nan Chinese.

9.2.2 BestAvailableLocale (*availableLocales*, *locale*)

The BestAvailableLocale abstract operation compares the provided argument *locale*, which must be a String value with a structurally valid and canonicalized BCP 47 language tag, against the locales in *availableLocales* and returns either the longest non-empty prefix of *locale* that is an element of *availableLocales*, or **undefined** if there is no such element. It uses the fallback mechanism of RFC 4647, section 3.4. The following steps are taken:

1. Let *candidate* be *locale*.
2. Repeat,

a. If *availableLocales* contains an element equal to *candidate*, return *candidate*.
b. Let *pos* be the character index of the last occurrence of **"-"** (U+002D) within *candidate*. If that character does not occur, return **undefined**.
c. If *pos* ≥ 2 and the character **"-"** occurs at index *pos*-2 of candidate, decrease *pos* by 2.
d. Let *candidate* be the substring of *candidate* from position 0, inclusive, to position *pos*, exclusive.

9.2.3 LookupMatcher (*availableLocales*, *requestedLocales*)

The LookupMatcher abstract operation compares *requestedLocales*, which must be a List as returned by CanonicalizeLocaleList, against the locales in *availableLocales* and determines the best available language to meet the request. The following steps are taken:

1. Let *result* be a new Record.
2. For each element *locale* of *requestedLocales* in List order, do
    a. Let *noExtensionsLocale* be the String value that is *locale* with all Unicode locale extension sequences removed.
    b. Let *availableLocale* be BestAvailableLocale(*availableLocales*, *noExtensionsLocale*).
    c. If *availableLocale* is not **undefined**, then
        i. Set *result*.[[locale]] to *availableLocale*.
        ii. If *locale* and *noExtensionsLocale* are not the same String value, then
            1. Let *extension* be the String value consisting of the first substring of *locale* that is a Unicode locale extension sequence.
            2. Set *result*.[[extension]] to *extension*.
        iii. Return *result*.
3. Let *defLocale* be DefaultLocale().
4. Set *result*.[[locale]] to *defLocale*.
5. Return *result*.

NOTE
The algorithm is based on the Lookup algorithm described in RFC 4647 section 3.4, but options specified through Unicode locale extension sequences are ignored in the lookup. Information about such subsequences is returned separately. The abstract operation returns a record with a [[locale]] field, whose value is the language tag of the selected locale, which must be an element of *availableLocales*. If the language tag of the request locale that led to the selected locale contained a Unicode locale extension sequence, then the returned record also contains an [[extension]] field whose value is the first Unicode locale extension sequence within the request locale language tag.

9.2.4 BestFitMatcher (*availableLocales*, *requestedLocales*)

The BestFitMatcher abstract operation compares *requestedLocales*, which must be a List as returned by CanonicalizeLocaleList, against the locales in *availableLocales* and determines the best available language to meet the request. The algorithm is implementation dependent, but should produce results that a typical user of the requested locales would perceive as at least as good as those produced by the LookupMatcher abstract operation. Options specified through Unicode locale extension sequences must be ignored by the algorithm. Information about such subsequences is returned separately. The abstract operation returns a record with a [[locale]] field, whose value is the language tag of the selected locale, which must be an element of *availableLocales*. If the language tag of the request locale that led to the selected locale contained a Unicode locale extension sequence, then the returned record also contains an [[extension]] field whose value is the first

Unicode locale extension sequence within the request locale language tag.

9.2.5 UnicodeExtensionValue (*extension*, *key*)

The abstract operation UnicodeExtensionValue is called with *extension*, which must be a Unicode locale extension sequence, and String *key*. This operation returns the type subtags for *key* by performing the following steps:

1. Assert: The number of elements in *key* is 2.
2. Let *size* be the number of elements in *extension*.
3. Let *searchValue* be the concatenation of **"-"**, *key*, and **"-"**.
4. Let *pos* be Call(%StringProto_indexOf%, *extension*, « *searchValue* »).
5. If *pos* ≠ -1, then
    a. Let *start* be *pos* + 4.
    b. Let *end* be *start*.
    c. Let *k* be *start*.
    d. Let *done* be **false**.
    e. Repeat, while *done* is **false**
        i. Let *e* be Call(%StringProto_indexOf%, *extension*, « **"-"**, *k* »).
        ii. If *e* = -1, let *len* be *size* - *k*; else let *len* be *e* - *k*.
        iii. If *len* = 2, then
            1. Let *done* be **true**.
        iv. Else if *e* = -1, then
            1. Let *end* be *size*.
            2. Let *done* be **true**.
        v. Else,
            1. Let *end* be *e*.
            2. Let *k* be *e* + 1.
    f. Return the String value equal to the substring of *extension* consisting of the code units at indices *start* (inclusive) through *end* (exclusive).
6. Let *searchValue* be the concatenation of **"-"** and *key*.
7. Let *pos* be Call(%StringProto_indexOf%, *extension*, « *searchValue* »).
8. If *pos* ≠ -1 and *pos* + 3 = *size*, then
    a. Return the empty String.
9. Return **undefined**.

NOTE

Non-normative summary: UnicodeExtensionValue returns the type subtags of the first keyword for a given key. For example, UnicodeExtensionValue(**"u-ca-ethiopic-amete-alem-ca-ethioaa"**, **"ca"**) returns **"ethiopic-amete-alem"**. If the keyword for *key* has no type subtags, UnicodeExtensionValue returns the empty String. If *extension* contains no keyword for *key*, **undefined** is returned.

9.2.6 ResolveLocale (*availableLocales*, *requestedLocales*, *options*, *relevantExtensionKeys*, *localeData*)

The ResolveLocale abstract operation compares a BCP 47 language priority list *requestedLocales* against the locales in

availableLocales and determines the best available language to meet the request. *availableLocales*, *requestedLocales*, and *relevantExtensionKeys* must be provided as List values, *options* and *localeData* as Records.

The following steps are taken:

1. Let *matcher* be *options*.[[localeMatcher]].
2. If *matcher* is **"lookup"**, then
    a. Let *r* be LookupMatcher(*availableLocales*, *requestedLocales*).
3. Else,
    a. Let *r* be BestFitMatcher(*availableLocales*, *requestedLocales*).
4. Let *foundLocale* be *r*.[[locale]].
5. Let *result* be a new Record.
6. Set *result*.[[dataLocale]] to *foundLocale*.
7. Let *supportedExtension* be **"-u"**.
8. For each element *key* of *relevantExtensionKeys* in List order, do
    a. Let *foundLocaleData* be *localeData*.[[<*foundLocale*>]].
    b. Assert: Type(*foundLocaleData*) is Record.
    c. Let *keyLocaleData* be *foundLocaleData*.[[<*key*>]].
    d. Assert: Type(*keyLocaleData*) is List.
    e. Let *value* be *keyLocaleData*[0].
    f. Assert: Type(*value*) is either String or Null.
    g. Let *supportedExtensionAddition* be **""**.
    h. If *r* has an [[extension]] field, then
        i. Let *requestedValue* be UnicodeExtensionValue(*r*.[[extension]], *key*).
        ii. If *requestedValue* is not **undefined**, then
            1. If *requestedValue* is not the empty String, then
                a. If *keyLocaleData* contains *requestedValue*, then
                    i. Let *value* be *requestedValue*.
                    ii. Let *supportedExtensionAddition* be the concatenation of **"-"**, *key*, **"-"**, and *value*.
            2. Else if *keyLocaleData* contains **"true"**, then
                a. Let *value* be **"true"**.
                b. Let *supportedExtensionAddition* be the concatenation of **"-"** and *key*.
    i. If *options* has a field [[<*key*>]], then
        i. Let *optionsValue* be *options*.[[<*key*>]].
        ii. Assert: Type(*optionsValue*) is either String, Undefined, or Null.
        iii. If *keyLocaleData* contains *optionsValue*, then
            1. If SameValue(*optionsValue*, *value*) is **false**, then
                a. Let *value* be *optionsValue*.
                b. Let *supportedExtensionAddition* be **""**.
    j. Set *result*.[[<*key*>]] to *value*.
    k. Append *supportedExtensionAddition* to *supportedExtension*.
9. If the number of elements in *supportedExtension* is greater than 2, then
    a. Let *privateIndex* be Call(%StringProto_indexOf%, *foundLocale*, « **"-x-"** »).
    b. If *privateIndex* = -1, then
        i. Let *foundLocale* be the concatenation of *foundLocale* and *supportedExtension*.