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*.
    c. Else,
        i. Let *preExtension* be the substring of *foundLocale* from position 0, inclusive, to position *privateIndex*, exclusive.
        ii. Let *postExtension* be the substring of *foundLocale* from position *privateIndex* to the end of the string.
        iii. Let *foundLocale* be the concatenation of *preExtension*, *supportedExtension*, and *postExtension*.
    d. Assert: IsStructurallyValidLanguageTag(*foundLocale*) is **true**.
    e. Let *foundLocale* be CanonicalizeLanguageTag(*foundLocale*).
10. Set *result*.[[locale]] to *foundLocale*.
11. Return *result*.

NOTE

Non-normative summary: Two algorithms are available to match the locales: the Lookup algorithm described in RFC 4647 section 3.4, and an implementation dependent best-fit algorithm. Independent of the locale matching algorithm, options specified through Unicode locale extension sequences are negotiated separately, taking the caller's relevant extension keys and locale data as well as client-provided options into consideration. The abstract operation returns a record with a [[locale]] field whose value is the language tag of the selected locale, and fields for each key in *relevantExtensionKeys* providing the selected value for that key.

9.2.7 LookupSupportedLocales (*availableLocales*, *requestedLocales*)

The LookupSupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale when using the BCP 47 Lookup algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The following steps are taken:

1. Let *subset* be a new empty List.

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**, append *locale* to the end of *subset*.
3. Return *subset*.

9.2.8 BestFitSupportedLocales (*availableLocales*, *requestedLocales*)

The BestFitSupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale when using the Best Fit Matcher algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The steps taken are implementation dependent.

9.2.9 SupportedLocales (*availableLocales*, *requestedLocales*, *options*)

The SupportedLocales abstract operation returns the subset of the provided BCP 47 language priority list *requestedLocales* for which *availableLocales* has a matching locale. Two algorithms are available to match the locales: the Lookup algorithm described in RFC 4647 section 3.4, and an implementation dependent best-fit algorithm. Locales appear in the same order in the returned list as in *requestedLocales*. The following steps are taken:

1. If *options* is not **undefined**, then
    a. Let *options* be ? ToObject(*options*).
    b. Let *matcher* be ? GetOption(*options*, `"localeMatcher"`, `"string"`, « `"lookup"`, `"best fit"` », `"best fit"`).
2. Else, let *matcher* be `"best fit"`.
3. If *matcher* is `"best fit"`, then
    a. Let *supportedLocales* be BestFitSupportedLocales(*availableLocales*, *requestedLocales*).
4. Else,
    a. Let *supportedLocales* be LookupSupportedLocales(*availableLocales*, *requestedLocales*).
5. Return CreateArrayFromList(*supportedLocales*).

9.2.10 GetOption (*options*, *property*, *type*, *values*, *fallback*)

The abstract operation GetOption extracts the value of the property named *property* from the provided *options* object, converts it to the required *type*, checks whether it is one of a List of allowed *values*, and fills in a *fallback* value if necessary. If *values* is **undefined**, there is no fixed set of values and any is permitted.

1. Let *value* be ? Get(*options*, *property*).
2. If *value* is not **undefined**, then
    a. Assert: *type* is `"boolean"` or `"string"`.
    b. If *type* is `"boolean"`, then
        i. Let *value* be ToBoolean(*value*).
    c. If *type* is `"string"`, then
        i. Let *value* be ? ToString(*value*).
    d. If *values* is not **undefined**, then
        i. If *values* does not contain an element equal to *value*, throw a **RangeError** exception.
    e. Return *value*.
3. Else, return *fallback*.

9.2.11 DefaultNumberOption (*value*, *minimum*, *maximum*, *fallback*)

The abstract operation DefaultNumberOption converts *value* to a Number value, checks whether it is in the allowed range, and fills in a *fallback* value if necessary.

1. If *value* is not **undefined**, then
   a. Let *value* be ? ToNumber(*value*).
   b. If *value* is **NaN** or less than *minimum* or greater than *maximum*, throw a **RangeError** exception.
   c. Return floor(*value*).
2. Else, return *fallback*.

9.2.12 GetNumberOption (*options*, *property*, *minimum*, *maximum*, *fallback*)

The abstract operation GetNumberOption extracts the value of the property named *property* from the provided *options* object, converts it to a Number value, checks whether it is in the allowed range, and fills in a *fallback* value if necessary.

1. Let *value* be ? Get(*options*, *property*).
2. Return ? DefaultNumberOption(*value*, *minimum*, *maximum*, *fallback*).

10 Collator Objects

10.1 The Intl.Collator Constructor

The Intl.Collator constructor is the %Collator% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

10.1.1 InitializeCollator (*collator*, *locales*, *options*)

The abstract operation InitializeCollator accepts the arguments *collator* (which must be an object), *locales*, and *options*. It initializes *collator* as a **Collator** object. The following steps are taken:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. If *options* is **undefined**, then
   a. Let *options* be ObjectCreate(**null**).
3. Else,
   a. Let *options* be ? ToObject(*options*).
4. Let *usage* be ? GetOption(*options*, `"usage"`, `"string"`, « `"sort"`, `"search"` », `"sort"`).
5. Set *collator*.[[Usage]] to *usage*.
6. If *usage* is `"sort"`, then
   a. Let *localeData* be %Collator%.[[SortLocaleData]].
7. Else,
   a. Let *localeData* be %Collator%.[[SearchLocaleData]].
8. Let *opt* be a new Record.
9. Let *matcher* be ? GetOption(*options*, `"localeMatcher"`, `"string"`, « `"lookup"`, `"best fit"` », `"best fit"`).
10. Set *opt*.[[localeMatcher]] to *matcher*.

11. Let *numeric* be ? GetOption(*options*, `"numeric"`, `"boolean"`, **undefined**, **undefined**).
12. If *numeric* is not **undefined**, then
    a. Let *numeric* be ! ToString(*numeric*).
13. Set *opt*.[[kn]] to *numeric*.
14. Let *caseFirst* be ? GetOption(*options*, `"caseFirst"`, `"string"`, « `"upper"`, `"lower"`, `"false"` », **undefined**).
15. Set *opt*.[[kf]] to *caseFirst*.
16. Let *relevantExtensionKeys* be %Collator%.[[RelevantExtensionKeys]].
17. Let *r* be ResolveLocale(%Collator%.[[AvailableLocales]], *requestedLocales*, *opt*, *relevantExtensionKeys*, *localeData*).
18. Set *collator*.[[Locale]] to *r*.[[locale]].
19. Let *collation* be *r*.[[co]].
20. If *collation* is **null**, let *collation* be `"default"`.
21. Set *collator*.[[Collation]] to *collation*.
22. If *relevantExtensionKeys* contains `"kn"`, then
    a. Set *collator*.[[Numeric]] to ! SameValue(*r*.[[kn]], `"true"`).
23. If *relevantExtensionKeys* contains `"kf"`, then
    a. Set *collator*.[[CaseFirst]] to *r*.[[kf]].
24. Let *sensitivity* be ? GetOption(*options*, `"sensitivity"`, `"string"`, « `"base"`, `"accent"`, `"case"`, `"variant"` », **undefined**).
25. If *sensitivity* is **undefined**, then
    a. If *usage* is `"sort"`, then
        i. Let *sensitivity* be `"variant"`.
    b. Else,
        i. Let *dataLocale* be *r*.[[dataLocale]].
        ii. Let *dataLocaleData* be *localeData*.[[<*dataLocale*>]].
        iii. Let *sensitivity* be *dataLocaleData*.[[sensitivity]].
26. Set *collator*.[[Sensitivity]] to *sensitivity*.
27. Let *ignorePunctuation* be ? GetOption(*options*, `"ignorePunctuation"`, `"boolean"`, **undefined**, **false**).
28. Set *collator*.[[IgnorePunctuation]] to *ignorePunctuation*.
29. Return *collator*.

10.1.2 Intl.Collator ([*locales* [, *options*]])

When the `Intl.Collator` function is called with optional arguments *locales* and *options*, the following steps are taken:

1. If NewTarget is **undefined**, let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *internalSlotsList* be « [[InitializedCollator]], [[Locale]], [[Usage]], [[Sensitivity]], [[IgnorePunctuation]], [[Collation]], [[BoundCompare]] ».
3. If %Collator%.[[RelevantExtensionKeys]] contains `"kn"`, then
    a. Append [[Numeric]] as the last element of *internalSlotsList*.
4. If %Collator%.[[RelevantExtensionKeys]] contains `"kf"`, then
    a. Append [[CaseFirst]] as the last element of *internalSlotsList*.
5. Let *collator* be ? OrdinaryCreateFromConstructor(*newTarget*, `"%CollatorPrototype%"`, *internalSlotsList*).
6. Return ? InitializeCollator(*collator*, *locales*, *options*).

10.2 Properties of the Intl.Collator Constructor

The Intl.Collator constructor has the following properties:

10.2.1 Intl.Collator.prototype

The value of `Intl.Collator.prototype` is %CollatorPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

10.2.2 Intl.Collator.supportedLocalesOf (*locales* [, *options*])

When the `supportedLocalesOf` method is called, the following steps are taken:

1. Let *availableLocales* be %Collator%.[[AvailableLocales]].
2. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
3. Return ? SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the `length` property of the **supportedLocalesOf** method is 1.

10.2.3 Internal Slots

The value of the [[AvailableLocales]] internal slot is implementation defined within the constraints described in 9.1. The value of the [[RelevantExtensionKeys]] internal slot is a List that must include the element `"co"`, may include any or all of the elements `"kn"` and `"kf"`, and must not include any other elements.

NOTE
Unicode Technical Standard 35 describes ten locale extension keys that are relevant to collation: `"co"` for collator usage and specializations, `"ka"` for alternate handling, `"kb"` for backward second level weight, `"kc"` for case level, `"kn"` for numeric, `"kh"` for hiragana quaternary, `"kk"` for normalization, `"kf"` for case first, `"kr"` for reordering, `"ks"` for collation strength, and `"vt"` for variable top. Collator, however, requires that the usage is specified through the usage property of the options object, alternate handling through the ignorePunctuation property of the options object, and case level and the strength through the sensitivity property of the options object. The `"co"` key in the language tag is supported only for collator specializations, and the keys `"kb"`, `"kh"`, `"kk"`, `"kr"`, and `"vt"` are not allowed in this version of the Internationalization API. Support for the remaining keys is implementation dependent.

The values of the [[SortLocaleData]] and [[SearchLocaleData]] internal slots are implementation defined within the constraints described in 9.1 and the following additional constraints:

- The first element of [[SortLocaleData]][[<*locale*>]].[[co]] and [[SearchLocaleData]][[<*locale*>]].[[co]] must be **null** for all locale values.
- The values `"standard"` and `"search"` must not be used as elements in any [[SortLocaleData]].[[<*locale*>]].[[co]] and [[SearchLocaleData]].[[<*locale*>]].[[co]] list.
- [[SearchLocaleData]][[<*locale*>]] must have a sensitivity field with a String value equal to `"base"`, `"accent"`, `"case"`, or `"variant"` for all locale values.

10.3 Properties of the Intl.Collator Prototype Object

The Intl.Collator prototype object is itself an ordinary object. %CollatorPrototype% is not an Intl.Collator instance and does not have an [[InitializedCollator]] internal slot or any of the other internal slots of Intl.Collator instance objects.

10.3.1 Intl.Collator.prototype.constructor

The initial value of `Intl.Collator.prototype.constructor` is the intrinsic object %Collator%.

10.3.2 Intl.Collator.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

10.3.3 get Intl.Collator.prototype.compare

This named accessor property returns a function that compares two strings according to the sort order of this Collator object.

Intl.Collator.prototype.compare is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *collator* be **this** value.
2. If Type(*collator*) is not Object, throw a **TypeError** exception.
3. If *collator* does not have an [[InitializedCollator]] internal slot, throw a **TypeError** exception.
4. If *collator*.[[BoundCompare]] is **undefined**, then
   a. Let *F* be a new built-in function object as defined in 10.3.3.1.
   b. Set *F*.[[Collator]] to *collator*.
   c. Set *collator*.[[BoundCompare]] to *F*.
5. Return *collator*.[[BoundCompare]].

NOTE
The returned function is bound to *collator* so that it can be passed directly to `Array.prototype.sort` or other functions.

10.3.3.1 Collator Compare Functions

A Collator compare function is an anonymous built-in function that has a [[Collator]] internal slot.

When a Collator compare function *F* is called with arguments *x* and *y*, the following steps are taken:

1. Let *collator* be *F*.[[Collator]].
2. Assert: Type(*collator*) is Object and *collator* has an [[InitializedCollator]] internal slot.
3. If *x* is not provided, let *x* be **undefined**.
4. If *y* is not provided, let *y* be **undefined**.
5. Let *X* be ? ToString(*x*).
6. Let *Y* be ? ToString(*y*).
7. Return CompareStrings(*collator*, *X*, *Y*).

The `length` property of a Collator compare function is 2.

10.3.3.2 CompareStrings (*collator*, *x*, *y*)

When the CompareStrings abstract operation is called with arguments *collator* (which must be an object initialized as a Collator), *x* and *y* (which must be String values), it returns a Number other than **NaN** that represents the result of a locale-sensitive String comparison of *x* with *y*. The two Strings are compared in an implementation-defined fashion. The result is intended to order String values in the sort order specified by the effective locale and collation options computed during construction of *collator*, and will be negative, zero, or positive, depending on whether *x* comes before *y* in the sort order, the Strings are equal under the sort order, or *x* comes after *y* in the sort order, respectively. String values must be interpreted as UTF-16 code unit sequences, and a surrogate pair (a code unit in the range 0xD800 to 0xDBFF followed by a code unit in the range 0xDC00 to 0xDFFF) within a string must be interpreted as the corresponding code point.

The sensitivity of *collator* is interpreted as follows:

- base: Only strings that differ in base letters compare as unequal. Examples: a ≠ b, a = á, a = A.
- accent: Only strings that differ in base letters or accents and other diacritic marks compare as unequal. Examples: a ≠ b, a ≠ á, a = A.
- case: Only strings that differ in base letters or case compare as unequal. Examples: a ≠ b, a = á, a ≠ A.
- variant: Strings that differ in base letters, accents and other diacritic marks, or case compare as unequal. Other differences may also be taken into consideration. Examples: a ≠ b, a ≠ á, a ≠ A.

NOTE 1
In some languages, certain letters with diacritic marks are considered base letters. For example, in Swedish, "ö" is a base letter that's different from "o".

If the collator is set to ignore punctuation, then strings that differ only in punctuation compare as equal.

For the interpretation of options settable through extension keys, see Unicode Technical Standard 35.

The CompareStrings abstract operation with any given *collator* argument, if considered as a function of the remaining two arguments *x* and *y*, must be a consistent comparison function (as defined in ES2020, 22.1.3.25) on the set of all Strings.

The actual return values are implementation-defined to permit implementers to encode additional information in the value. The method is required to return **+0** when comparing Strings that are considered canonically equivalent by the Unicode standard.

NOTE 2
It is recommended that the CompareStrings abstract operation be implemented following Unicode Technical Standard 10, Unicode Collation Algorithm (available at https://unicode.org/reports/tr10/), using tailorings for the effective locale and collation options of *collator*. It is recommended that implementations use the tailorings provided by the Common Locale Data Repository (available at http://cldr.unicode.org).

NOTE 3
Applications should not assume that the behaviour of the CompareStrings abstract operation for Collator instances with the same resolved options will remain the same for different versions of the same implementation.

10.3.4 Intl.Collator.prototype.resolvedOptions ()

This function provides access to the locale and collation options computed during initialization of the object.

1. Let *collator* be **this** value.

2. If Type(*collator*) is not Object, throw a **TypeError** exception.
3. If *collator* does not have an [[InitializedCollator]] internal slot, throw a **TypeError** exception.
4. Let *options* be ! ObjectCreate(%ObjectPrototype%).
5. For each row of Table 2, except the header row, in table order, do
    a. Let *p* be the Property value of the current row.
    b. Let *v* be the value of *collator*'s internal slot whose name is the Internal Slot value of the current row.
    c. If the current row has an Extension Key value, then
        i. Let *extensionKey* be the Extension Key value of the current row.
        ii. If %Collator%.[[RelevantExtensionKeys]] does not contain *extensionKey*, then
            1. Let *v* be **undefined**.
    d. If *v* is not **undefined**, then
        i. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 2: Resolved Options of Collator Instances

Internal Slot	Property	Extension Key
[[Locale]]	`"locale"`	
[[Usage]]	`"usage"`	
[[Sensitivity]]	`"sensitivity"`	
[[IgnorePunctuation]]	`"ignorePunctuation"`	
[[Collation]]	`"collation"`	
[[Numeric]]	`"numeric"`	kn
[[CaseFirst]]	`"caseFirst"`	kf

10.4 Properties of Intl.Collator Instances

Intl.Collator instances are ordinary objects that inherit properties from %CollatorPrototype%.

Intl.Collator instances have an [[InitializedCollator]] internal slot.

Intl.Collator instances also have several internal slots that are computed by the constructor:

- [[Locale]] is a String value with the language tag of the locale whose localization is used for collation.
- [[Usage]] is one of the String values `"sort"` or `"search"`, identifying the collator usage.
- [[Sensitivity]] is one of the String values `"base"`, `"accent"`, `"case"`, or `"variant"`, identifying the collator's sensitivity.
- [[IgnorePunctuation]] is a Boolean value, specifying whether punctuation should be ignored in comparisons.
- [[Collation]] is a String value with the `"type"` given in Unicode Technical Standard 35 for the collation, except that the values `"standard"` and `"search"` are not allowed, while the value `"default"` is allowed.

Intl.Collator instances also have the following internal slots if the key corresponding to the name of the internal slot in Table 2 is included in the [[RelevantExtensionKeys]] internal slot of Intl.Collator:

- [[Numeric]] is a Boolean value, specifying whether numeric sorting is used.

- [[CaseFirst]] is one of the String values **"upper"**, **"lower"**, or **"false"**.

Finally, Intl.Collator instances have a [[BoundCompare]] internal slot that caches the function returned by the compare accessor (10.3.3).

11 NumberFormat Objects

11.1 Abstract Operations For NumberFormat Objects

11.1.1 SetNumberFormatDigitOptions (*intlObj*, *options*, *mnfdDefault*, *mxfdDefault*)

The abstract operation SetNumberFormatDigitOptions applies digit options used for number formatting onto the intl object.

1. Assert: Type(*intlObj*) is Object.
2. Assert: Type(*options*) is Object.
3. Assert: Type(*mnfdDefault*) is Number.
4. Assert: Type(*mxfdDefault*) is Number.
5. Let *mnid* be ? GetNumberOption(*options*, **"minimumIntegerDigits"**, 1, 21, 1).
6. Let *mnfd* be ? GetNumberOption(*options*, **"minimumFractionDigits"**, 0, 20, *mnfdDefault*).
7. Let *mxfdActualDefault* be max(*mnfd*, *mxfdDefault*).
8. Let *mxfd* be ? GetNumberOption(*options*, **"maximumFractionDigits"**, *mnfd*, 20, *mxfdActualDefault*).
9. Let *mnsd* be ? Get(*options*, **"minimumSignificantDigits"**).
10. Let *mxsd* be ? Get(*options*, **"maximumSignificantDigits"**).
11. Set *intlObj*.[[MinimumIntegerDigits]] to *mnid*.
12. Set *intlObj*.[[MinimumFractionDigits]] to *mnfd*.
13. Set *intlObj*.[[MaximumFractionDigits]] to *mxfd*.
14. If *mnsd* is not **undefined** or *mxsd* is not **undefined**, then
    a. Let *mnsd* be ? DefaultNumberOption(*mnsd*, 1, 21, 1).
    b. Let *mxsd* be ? DefaultNumberOption(*mxsd*, *mnsd*, 21, 21).
    c. Set *intlObj*.[[MinimumSignificantDigits]] to *mnsd*.
    d. Set *intlObj*.[[MaximumSignificantDigits]] to *mxsd*.

11.1.2 InitializeNumberFormat (*numberFormat*, *locales*, *options*)

The abstract operation InitializeNumberFormat accepts the arguments *numberFormat* (which must be an object), *locales*, and *options*. It initializes *numberFormat* as a NumberFormat object. The following steps are taken:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. If *options* is **undefined**, then
    a. Let *options* be ObjectCreate(**null**).
3. Else,
    a. Let *options* be ? ToObject(*options*).
4. Let *opt* be a new Record.
5. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, **"string"**, « **"lookup"**, **"best fit"** »,

`"best fit"`).

6. Set *opt*.[[localeMatcher]] to *matcher*.
7. Let *localeData* be %NumberFormat%.[[LocaleData]].
8. Let *r* be ResolveLocale(%NumberFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %NumberFormat%.[[RelevantExtensionKeys]], *localeData*).
9. Set *numberFormat*.[[Locale]] to *r*.[[locale]].
10. Set *numberFormat*.[[NumberingSystem]] to *r*.[[nu]].
11. Let *dataLocale* be *r*.[[dataLocale]].
12. Let *style* be ? GetOption(*options*, `"style"`, `"string"`, « `"decimal"`, `"percent"`, `"currency"` », `"decimal"`).
13. Set *numberFormat*.[[Style]] to *style*.
14. Let *currency* be ? GetOption(*options*, `"currency"`, `"string"`, **undefined**, **undefined**).
15. If *currency* is not **undefined**, then
    a. If the result of IsWellFormedCurrencyCode(*currency*) is **false**, throw a **RangeError** exception.
16. If *style* is `"currency"` and *currency* is **undefined**, throw a **TypeError** exception.
17. If *style* is `"currency"`, then
    a. Let *currency* be the result of converting *currency* to upper case as specified in 6.1.
    b. Set *numberFormat*.[[Currency]] to *currency*.
    c. Let *cDigits* be CurrencyDigits(*currency*).
18. Let *currencyDisplay* be ? GetOption(*options*, `"currencyDisplay"`, `"string"`, « `"code"`, `"symbol"`, `"name"` », `"symbol"`).
19. If *style* is `"currency"`, set *numberFormat*.[[CurrencyDisplay]] to *currencyDisplay*.
20. If *style* is `"currency"`, then
    a. Let *mnfdDefault* be *cDigits*.
    b. Let *mxfdDefault* be *cDigits*.
21. Else,
    a. Let *mnfdDefault* be 0.
    b. If *style* is `"percent"`, then
        i. Let *mxfdDefault* be 0.
    c. Else,
        i. Let *mxfdDefault* be 3.
22. Perform ? SetNumberFormatDigitOptions(*numberFormat*, *options*, *mnfdDefault*, *mxfdDefault*).
23. Let *useGrouping* be ? GetOption(*options*, `"useGrouping"`, `"boolean"`, **undefined**, **true**).
24. Set *numberFormat*.[[UseGrouping]] to *useGrouping*.
25. Let *dataLocaleData* be *localeData*.[[<*dataLocale*>]].
26. Let *patterns* be *dataLocaleData*.[[patterns]].
27. Assert: *patterns* is a record (see 11.3.3).
28. Let *stylePatterns* be *patterns*.[[<*style*>]].
29. Set *numberFormat*.[[PositivePattern]] to *stylePatterns*.[[positivePattern]].
30. Set *numberFormat*.[[NegativePattern]] to *stylePatterns*.[[negativePattern]].
31. Return *numberFormat*.

11.1.3 CurrencyDigits (*currency*)

When the abstract operation CurrencyDigits is called with an argument *currency* (which must be an upper case String value), the following steps are taken:

1. If the ISO 4217 currency and funds code list contains *currency* as an alphabetic code, return the minor unit value

corresponding to the *currency* from the list; otherwise, return 2.

11.1.4 Number Format Functions

A Number format function is an anonymous built-in function that has a [[NumberFormat]] internal slot.

When a Number format function *F* is called with optional argument *value*, the following steps are taken:

1. Let *nf* be *F*.[[NumberFormat]].
2. Assert: Type(*nf*) is Object and *nf* has an [[InitializedNumberFormat]] internal slot.
3. If *value* is not provided, let *value* be **undefined**.
4. Let *x* be ? ToNumeric(*value*).
5. Return FormatNumeric(*nf*, *x*).

The `length` property of a Number format function is 1.

11.1.5 FormatNumericToString (*intlObject*, *x*)

The FormatNumericToString abstract operation is called with arguments *intlObject* (which must be an object with [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], and [[MaximumFractionDigits]] internal slots), and *x* (which must be a Number or BigInt value), and returns *x* as a string value with digits formatted according to the five formatting parameters.

1. If *intlObject*.[[MinimumSignificantDigits]] and *intlObject*.[[MaximumSignificantDigits]] are both not **undefined**, then
    a. Let *result* be ToRawPrecision(*x*, *intlObject*.[[MinimumSignificantDigits]], *intlObject*.[[MaximumSignificantDigits]]).
2. Else,
    a. Let *result* be ToRawFixed(*x*, *intlObject*.[[MinimumIntegerDigits]], *intlObject*.[[MinimumFractionDigits]], *intlObject*.[[MaximumFractionDigits]]).
3. Return *result*.

11.1.6 PartitionNumberPattern (*numberFormat*, *x*)

The PartitionNumberPattern abstract operation is called with arguments *numberFormat* (which must be an object initialized as a NumberFormat) and *x* (which must be a Number or BigInt value), interprets *x* as a numeric value, and creates the corresponding parts according to the effective locale and the formatting options of *numberFormat*. The following steps are taken:

1. If *x* is not **NaN** and *x* < 0 or *x* is **-0**, then
    a. Let *x* be -*x*.
    b. Let *pattern* be *numberFormat*.[[NegativePattern]].
2. Else,
    a. Let *pattern* be *numberFormat*.[[PositivePattern]].
3. Let *result* be a new empty List.
4. Let *beginIndex* be Call(%StringProto_indexOf%, *pattern*, « `"{"`, 0 »).
5. Let *endIndex* be 0.
6. Let *nextIndex* be 0.
7. Let *length* be the number of code units in *pattern*.

8. Repeat, while *beginIndex* is an integer index into *pattern*
    a. Set *endIndex* to Call(%StringProto_indexOf%, *pattern*, « `"}"`, *beginIndex* »).
    b. Assert: *endIndex* is greater than *beginIndex*.
    c. If *beginIndex* is greater than *nextIndex*, then
        i. Let *literal* be a substring of *pattern* from position *nextIndex*, inclusive, to position *beginIndex*, exclusive.
        ii. Append a new Record { [[Type]]: **`"literal"`**, [[Value]]: *literal* } as the last element of *result*.
    d. Let *p* be the substring of *pattern* from position *beginIndex*, exclusive, to position *endIndex*, exclusive.
    e. If *p* is equal to **`"number"`**, then
        i. If *x* is **NaN**, then
            1. Let *n* be an implementation- and locale-dependent (ILD) String value indicating the **NaN** value.
            2. Append a new Record { [[Type]]: **`"nan"`**, [[Value]]: *n* } as the last element of *result*.
        ii. Else if *x* is not a finite Number or BigInt,
            1. Let *n* be an ILD String value indicating infinity.
            2. Append a new Record { [[Type]]: **`"infinity"`**, [[Value]]: *n* } as the last element of *result*.
        iii. Else,
            1. If *numberFormat*.[[Style]] is **`"percent"`**, let *x* be 100 × *x*.
            2. Let *n* be FormatNumericToString(*numberFormat*, *x*).
            3. If the *numberFormat*.[[NumberingSystem]] matches one of the values in the **`"Numbering System"`** column of Table 3 below, then
                a. Let *digits* be a List whose 10 String valued elements are the UTF-16 string representations of the 10 *digits* specified in the **`"Digits"`** column of the matching row in Table 3.
                b. Replace each *digit* in *n* with the value of *digits*[*digit*].
            4. Else use an implementation dependent algorithm to map *n* to the appropriate representation of *n* in the given numbering system.
            5. Let *decimalSepIndex* be Call(%StringProto_indexOf%, *n*, « `"."`, 0 »).
            6. If *decimalSepIndex* > 0, then
                a. Let *integer* be the substring of *n* from position 0, inclusive, to position *decimalSepIndex*, exclusive.
                b. Let *fraction* be the substring of *n* from position *decimalSepIndex*, exclusive, to the end of *n*.
            7. Else,
                a. Let *integer* be *n*.
                b. Let *fraction* be **undefined**.
            8. If the *numberFormat*.[[UseGrouping]] is **true**, then
                a. Let *groupSepSymbol* be the implementation-, locale-, and numbering system-dependent (ILND) String representing the grouping separator.
                b. Let *groups* be a List whose elements are, in left to right order, the substrings defined by ILND set of locations within the *integer*.
                c. Assert: The number of elements in *groups* List is greater than 0.
                d. Repeat, while *groups* List is not empty
                    i. Remove the first element from *groups* and let *integerGroup* be the value of that element.
                    ii. Append a new Record { [[Type]]: **`"integer"`**, [[Value]]: *integerGroup* } as the last element of *result*.
                    iii. If *groups* List is not empty, then
                        i. Append a new Record { [[Type]]: **`"group"`**, [[Value]]: *groupSepSymbol* } as the last element of *result*.
            9. Else,
                a. Append a new Record { [[Type]]: **`"integer"`**, [[Value]]: *integer* } as the last element of

            result.
        10. If *fraction* is not **undefined**, then
            a. Let *decimalSepSymbol* be the ILND String representing the decimal separator.
            b. Append a new Record { [[Type]]: **`"decimal"`**, [[Value]]: *decimalSepSymbol* } as the last element of *result*.
            c. Append a new Record { [[Type]]: **`"fraction"`**, [[Value]]: *fraction* } as the last element of *result*.
    f. Else if *p* is equal to **`"plusSign"`**, then
        i. Let *plusSignSymbol* be the ILND String representing the plus sign.
        ii. Append a new Record { [[Type]]: **`"plusSign"`**, [[Value]]: *plusSignSymbol* } as the last element of *result*.
    g. Else if *p* is equal to **`"minusSign"`**, then
        i. Let *minusSignSymbol* be the ILND String representing the minus sign.
        ii. Append a new Record { [[Type]]: **`"minusSign"`**, [[Value]]: *minusSignSymbol* } as the last element of *result*.
    h. Else if *p* is equal to **`"percentSign"`** and *numberFormat*.[[Style]] is **`"percent"`**, then
        i. Let *percentSignSymbol* be the ILND String representing the percent sign.
        ii. Append a new Record { [[Type]]: **`"percentSign"`**, [[Value]]: *percentSignSymbol* } as the last element of *result*.
    i. Else if *p* is equal to **`"currency"`** and *numberFormat*.[[Style]] is **`"currency"`**, then
        i. Let *currency* be *numberFormat*.[[Currency]].
        ii. Assert: *numberFormat*.[[CurrencyDisplay]] is **`"code"`**, **`"symbol"`** or **`"name"`**.
        iii. If *numberFormat*.[[CurrencyDisplay]] is **`"code"`**, then
            1. Let *cd* be *currency*.
        iv. Else if *numberFormat*.[[CurrencyDisplay]] is **`"symbol"`**, then
            1. Let *cd* be an ILD string representing *currency* in short form. If the implementation does not have such a representation of *currency*, use *currency* itself.
        v. Else if *numberFormat*.[[CurrencyDisplay]] is **`"name"`**, then
            1. Let *cd* be an ILD string representing *currency* in long form. If the implementation does not have such a representation of *currency*, use *currency* itself.
        vi. Append a new Record { [[Type]]: **`"currency"`**, [[Value]]: *cd* } as the last element of *result*.
    j. Else,
        i. Let *unknown* be an ILND String based on *x* and *p*.
        ii. Append a new Record { [[Type]]: **`"unknown"`**, [[Value]]: *unknown* } as the last element of *result*.
    k. Set *nextIndex* to *endIndex* + 1.
    l. Set *beginIndex* to Call(%StringProto_indexOf%, *pattern*, « **`"{"`**, *nextIndex* »).
9. If *nextIndex* is less than *length*, then
    a. Let *literal* be the substring of *pattern* from position *nextIndex*, inclusive, to position *length*, exclusive.
    b. Append a new Record { [[Type]]: **`"literal"`**, [[Value]]: *literal* } as the last element of *result*.
10. Return *result*.

Table 3: Numbering systems with simple digit mappings

Numbering System	Digits
arab	U+0660 to U+0669
arabext	U+06F0 to U+06F9

Numbering System	Digits
bali	U+1B50 to U+1B59
beng	U+09E6 to U+09EF
deva	U+0966 to U+096F
fullwide	U+FF10 to U+FF19
gujr	U+0AE6 to U+0AEF
guru	U+0A66 to U+0A6F
hanidec	U+3007, U+4E00, U+4E8C, U+4E09, U+56DB, U+4E94, U+516D, U+4E03, U+516B, U+4E5D
khmr	U+17E0 to U+17E9
knda	U+0CE6 to U+0CEF
laoo	U+0ED0 to U+0ED9
latn	U+0030 to U+0039
limb	U+1946 to U+194F
mlym	U+0D66 to U+0D6F
mong	U+1810 to U+1819
mymr	U+1040 to U+1049
orya	U+0B66 to U+0B6F
tamldec	U+0BE6 to U+0BEF
telu	U+0C66 to U+0C6F
thai	U+0E50 to U+0E59
tibt	U+0F20 to U+0F29

NOTE 1
The computations rely on String values and locations within numeric strings that are dependent upon the implementation and the effective locale of *numberFormat* ("ILD") or upon the implementation, the effective locale, and the numbering system of *numberFormat* ("ILND"). The ILD and ILND Strings mentioned, other than those for currency names, must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.

NOTE 2
It is recommended that implementations use the locale provided by the Common Locale Data Repository (available at http://cldr.unicode.org).

11.1.7 FormatNumeric(*numberFormat*, *x*)

The FormatNumeric abstract operation is called with arguments *numberFormat* (which must be an object initialized as a NumberFormat) and *x* (which must be a Number or BigInt value), and performs the following steps:

1. Let *parts* be ? PartitionNumberPattern(*numberFormat*, *x*).
2. Let *result* be the empty String.
3. For each *part* in *parts*, do
   a. Set *result* to a String value produced by concatenating *result* and *part*.[[Value]].
4. Return *result*.

11.1.8 FormatNumericToParts(*numberFormat*, *x*)

The FormatNumericToParts abstract operation is called with arguments *numberFormat* (which must be an object initialized as a NumberFormat) and *x* (which must be a Number or BigInt value), and performs the following steps:

1. Let *parts* be ? PartitionNumberPattern(*numberFormat*, *x*).
2. Let *result* be ArrayCreate(0).
3. Let *n* be 0.
4. For each *part* in *parts*, do
   a. Let *O* be ObjectCreate(%ObjectPrototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, `"type"`, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, `"value"`, *part*.[[Value]]).
   d. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString(*n*), *O*).
   e. Increment *n* by 1.
5. Return *result*.

11.1.9 ToRawPrecision(*x*, *minPrecision*, *maxPrecision*)

When the ToRawPrecision abstract operation is called with arguments *x* (which must be a finite non-negative Number or BigInt), *minPrecision*, and *maxPrecision* (both must be integers between 1 and 21), the following steps are taken:

1. Let *p* be *maxPrecision*.
2. If *x* = 0, then
   a. Let *m* be the String consisting of *p* occurrences of the character `"0"`.
   b. Let *e* be 0.
3. Else,
   a. Let *e* and *n* be integers such that $10^{p-1} \le n < 10^{p}$ and for which the exact mathematical value of $n \times 10^{e-p+1} - x$ is as close to zero as possible. If there are two such sets of *e* and *n*, pick the *e* and *n* for which $n \times 10^{e-p+1}$ is larger.
   b. Let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If *e* ≥ *p*, then
   a. Return the concatenation of *m* and *e*-*p*+1 occurrences of the character `"0"`.
5. If *e* = *p*-1, then
   a. Return *m*.
6. If *e* ≥ 0, then
   a. Let *m* be the concatenation of the first *e*+1 characters of *m*, the character `"."`, and the remaining *p*–(*e*+1) characters of *m*.
7. If *e* < 0, then

   a. Let *m* be the concatenation of the String `"0."`, –(*e*+1) occurrences of the character `"0"`, and the string *m*.
8. If *m* contains the character `"."`, and *maxPrecision* > *minPrecision*, then
   a. Let *cut* be *maxPrecision* – *minPrecision*.
   b. Repeat, while *cut* > 0 and the last character of *m* is `"0"`
      i. Remove the last character from *m*.
      ii. Decrease *cut* by 1.
   c. If the last character of *m* is `"."`, then
      i. Remove the last character from *m*.
9. Return *m*.

11.1.10 ToRawFixed(*x*, *minInteger*, *minFraction*, *maxFraction*)

When the ToRawFixed abstract operation is called with arguments *x* (which must be a finite non-negative Number or BigInt), *minInteger* (which must be an integer between 1 and 21), *minFraction*, and *maxFraction* (which must be integers between 0 and 20), the following steps are taken:

1. Let *f* be *maxFraction*.
2. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such *n*, pick the larger *n*.
3. If *n* = 0, let *m* be the String `"0"`. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If *f* ≠ 0, then
   a. Let *k* be the number of characters in *m*.
   b. If *k* ≤ *f*, then
      i. Let *z* be the String consisting of *f*+1–*k* occurrences of the character `"0"`.
      ii. Let *m* be the concatenation of Strings *z* and *m*.
      iii. Let *k* be *f*+1.
   c. Let *a* be the first *k*–*f* characters of *m*, and let *b* be the remaining *f* characters of *m*.
   d. Let *m* be the concatenation of the three Strings *a*, `"."`, and *b*.
   e. Let *int* be the number of characters in *a*.
5. Else, let *int* be the number of characters in *m*.
6. Let *cut* be *maxFraction* – *minFraction*.
7. Repeat, while *cut* > 0 and the last character of *m* is `"0"`
   a. Remove the last character from *m*.
   b. Decrease *cut* by 1.
8. If the last character of *m* is `"."`, then
   a. Remove the last character from *m*.
9. If *int* < *minInteger*, then
   a. Let *z* be the String consisting of *minInteger*–*int* occurrences of the character `"0"`.
   b. Let *m* be the concatenation of Strings *z* and *m*.
10. Return *m*.

11.1.11 UnwrapNumberFormat(*nf*)

The UnwrapNumberFormat abstract operation gets the underlying NumberFormat operation for various methods which implement ECMA-402 v1 semantics for supporting initializing existing Intl objects.

1. Assert: Type(*nf*) is Object.

2. If *nf* does not have an [[InitializedNumberFormat]] internal slot and ? InstanceofOperator(*nf*, %NumberFormat%) is **true**, then
   a. Let *nf* be ? Get(*nf*, %Intl%.[[FallbackSymbol]]).
3. If Type(*nf*) is not Object or *nf* does not have an [[InitializedNumberFormat]] internal slot, then
   a. Throw a **TypeError** exception.
4. Return *nf*.

NOTE
See 8.1 Note 1 for the motivation of the normative optional text.

11.2 The Intl.NumberFormat Constructor

The NumberFormat constructor is the %NumberFormat% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

11.2.1 Intl.NumberFormat ([*locales* [, *options*]])

When the `Intl.NumberFormat` function is called with optional arguments *locales* and *options*, the following steps are taken:

1. If NewTarget is **undefined**, let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *numberFormat* be ? OrdinaryCreateFromConstructor(*newTarget*, `"%NumberFormatPrototype%"`, « [[InitializedNumberFormat]], [[Locale]], [[NumberingSystem]], [[Style]], [[Currency]], [[CurrencyDisplay]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[UseGrouping]], [[PositivePattern]], [[NegativePattern]], [[BoundFormat]] »).
3. Perform ? InitializeNumberFormat(*numberFormat*, *locales*, *options*).
4. Let *this* be the **this** value.
5. If NewTarget is **undefined** and ? InstanceofOperator(*this*, %NumberFormat%) is **true**, then
   a. Perform ? DefinePropertyOrThrow(*this*, %Intl%.[[FallbackSymbol]], PropertyDescriptor{ [[Value]]: *numberFormat*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
   b. Return *this*.
6. Return *numberFormat*.

NOTE
See 8.1 Note 1 for the motivation of the normative optional text.

11.3 Properties of the Intl.NumberFormat Constructor

The Intl.NumberFormat constructor has the following properties:

11.3.1 Intl.NumberFormat.prototype

The value of `Intl.NumberFormat.prototype` is %NumberFormatPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

11.3.2 Intl.NumberFormat.supportedLocalesOf (*locales* [, *options*])

When the `supportedLocalesOf` method is called with arguments *locales* and *options*, the following steps are taken:

1. Let *availableLocales* be %NumberFormat%.[[AvailableLocales]].
2. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
3. Return ? SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the `length` property of the **supportedLocalesOf** method is 1.

11.3.3 Internal slots

The value of the [[AvailableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[RelevantExtensionKeys]] internal slot is « `"nu"` ».

NOTE 1
Unicode Technical Standard 35 describes two locale extension keys that are relevant to number formatting, `"nu"` for numbering system and `"cu"` for currency. Intl.NumberFormat, however, requires that the currency of a currency format is specified through the currency property in the options objects.

The value of the [[LocaleData]] internal slot is implementation defined within the constraints described in 9.1 and the following additional constraints:

> The list that is the value of the `"nu"` field of any locale field of [[LocaleData]] must not include the values `"native"`, `"traditio"`, or `"finance"`.
> [[LocaleData]].[[<*locale*>]] must have a patterns field for all locale values *locale*. The value of this field must be a record, which must have fields with the names of the three number format styles: `"decimal"`, `"percent"`, and `"currency"`. Each of these fields in turn must be a record with the fields positivePattern and negativePattern. The value of these fields must be string values that must contain the substring `"{number}"` and may contain the substrings `"{plusSign}"`, and `"{minusSign}"`; the values within the percent field must also contain the substring `"{percentSign}"`; the values within the currency field must also contain the substring `"{currency}"`. The pattern strings must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.

NOTE 2
It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at http://cldr.unicode.org).

11.4 Properties of the Intl.NumberFormat Prototype Object

The Intl.NumberFormat prototype object is itself an ordinary object. %NumberFormatPrototype% is not an Intl.NumberFormat instance and does not have an [[InitializedNumberFormat]] internal slot or any of the other internal slots of Intl.NumberFormat instance objects.

11.4.1 Intl.NumberFormat.prototype.constructor

The initial value of `Intl.NumberFormat.prototype.constructor` is the intrinsic object %NumberFormat%.

11.4.2 Intl.NumberFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

11.4.3 get Intl.NumberFormat.prototype.format

Intl.NumberFormat.prototype.format is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *nf* be the **this** value.
2. If Type(*nf*) is not Object, throw a **TypeError** exception.
3. Let *nf* be ? UnwrapNumberFormat(*nf*).
4. If *nf*.[[BoundFormat]] is **undefined**, then
    a. Let *F* be a new built-in function object as defined in Number Format Functions (11.1.4).
    b. Set *F*.[[NumberFormat]] to *nf*.
    c. Set *nf*.[[BoundFormat]] to *F*.
5. Return *nf*.[[BoundFormat]].

NOTE
The returned function is bound to *nf* so that it can be passed directly to `Array.prototype.map` or other functions. This is considered a historical artefact, as part of a convention which is no longer followed for new features, but is preserved to maintain compatibility with existing programs.

11.4.4 Intl.NumberFormat.prototype.formatToParts (*value*)

When the `formatToParts` method is called with an optional argument *value*, the following steps are taken:

1. Let *nf* be the **this** value.
2. If Type(*nf*) is not Object, throw a **TypeError** exception.
3. If *nf* does not have an [[InitializedNumberFormat]] internal slot, throw a **TypeError** exception.
4. Let *x* be ? ToNumeric(*value*).
5. Return ? FormatNumericToParts(*nf*, *x*).

11.4.5 Intl.NumberFormat.prototype.resolvedOptions ()

This function provides access to the locale and formatting options computed during initialization of the object.

1. Let *nf* be **this** value.
2. If Type(*nf*) is not Object, throw a **TypeError** exception.
3. Let *nf* be ? UnwrapNumberFormat(*nf*).

4. Let *options* be ! ObjectCreate(%ObjectPrototype%).
5. For each row of Table 4, except the header row, in table order, do
    a. Let *p* be the Property value of the current row.
    b. Let *v* be the value of *nf*'s internal slot whose name is the Internal Slot value of the current row.
    c. If *v* is not **undefined**, then
        i. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 4: Resolved Options of NumberFormat Instances

Internal Slot	Property
[[Locale]]	**`"locale"`**
[[NumberingSystem]]	**`"numberingSystem"`**
[[Style]]	**`"style"`**
[[Currency]]	**`"currency"`**
[[CurrencyDisplay]]	**`"currencyDisplay"`**
[[MinimumIntegerDigits]]	**`"minimumIntegerDigits"`**
[[MinimumFractionDigits]]	**`"minimumFractionDigits"`**
[[MaximumFractionDigits]]	**`"maximumFractionDigits"`**
[[MinimumSignificantDigits]]	**`"minimumSignificantDigits"`**
[[MaximumSignificantDigits]]	**`"maximumSignificantDigits"`**
[[UseGrouping]]	**`"useGrouping"`**

11.5 Properties of Intl.NumberFormat Instances

Intl.NumberFormat instances inherit properties from %NumberFormatPrototype%.

Intl.NumberFormat instances have an [[InitializedNumberFormat]] internal slot.

Intl.NumberFormat instances also have several internal slots that are computed by the constructor:

[[Locale]] is a String value with the language tag of the locale whose localization is used for formatting.
[[NumberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
[[Style]] is one of the String values **`"decimal"`**, **`"currency"`**, or **`"percent"`**, identifying the number format style used.
[[Currency]] is a String value with the currency code identifying the currency to be used if formatting with the **`"currency"`** style. It is only used when [[Style]] has the value **`"currency"`**.
[[CurrencyDisplay]] is one of the String values **`"code"`**, **`"symbol"`**, or **`"name"`**, specifying whether to display the currency as an ISO 4217 alphabetic currency code, a localized currency symbol, or a localized currency name if formatting with the **`"currency"`** style. It is only used when [[Style]] has the value **`"currency"`**.
[[MinimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used.

Numbers will be padded with leading zeroes if necessary.

[[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.

[[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be shown. Either none or both of these properties are present; if they are, they override minimum and maximum integer and fraction digits – the formatter uses however many integer and fraction digits are required to display the specified number of significant digits.

[[UseGrouping]] is a Boolean value indicating whether a grouping separator should be used.

[[PositivePattern]] and [[NegativePattern]] are String values as described in 11.3.3.

Finally, Intl.NumberFormat instances have a [[BoundFormat]] internal slot that caches the function returned by the format accessor (11.4.3).

12 DateTimeFormat Objects

12.1 Abstract Operations For DateTimeFormat Objects

Several DateTimeFormat algorithms use values from the following table, which provides internal slots, property names and allowable values for the components of date and time formats:

Table 5: Components of date and time formats

Internal Slot	Property	Values
[[Weekday]]	`"weekday"`	`"narrow"`, `"short"`, `"long"`
[[Era]]	`"era"`	`"narrow"`, `"short"`, `"long"`
[[Year]]	`"year"`	`"2-digit"`, `"numeric"`
[[Month]]	`"month"`	`"2-digit"`, `"numeric"`, `"narrow"`, `"short"`, `"long"`
[[Day]]	`"day"`	`"2-digit"`, `"numeric"`
[[Hour]]	`"hour"`	`"2-digit"`, `"numeric"`
[[Minute]]	`"minute"`	`"2-digit"`, `"numeric"`
[[Second]]	`"second"`	`"2-digit"`, `"numeric"`
[[TimeZoneName]]	`"timeZoneName"`	`"short"`, `"long"`

12.1.1 InitializeDateTimeFormat (*dateTimeFormat*, *locales*, *options*)

The abstract operation InitializeDateTimeFormat accepts the arguments *dateTimeFormat* (which must be an object), *locales*, and *options*. It initializes *dateTimeFormat* as a DateTimeFormat object. This abstract operation functions as follows:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).

2. Let *options* be ? ToDateTimeOptions(*options*, **"any"**, **"date"**).
3. Let *opt* be a new Record.
4. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, **"string"**, « **"lookup"**, **"best fit"** », **"best fit"**).
5. Set *opt*.[[localeMatcher]] to *matcher*.
6. Let *hour12* be ? GetOption(*options*, **"hour12"**, **"boolean"**, **undefined**, **undefined**).
7. Let *hourCycle* be ? GetOption(*options*, **"hourCycle"**, **"string"**, « **"h11"**, **"h12"**, **"h23"**, **"h24"** », **undefined**).
8. If *hour12* is not **undefined**, then
    a. Let *hourCycle* be **null**.
9. Set *opt*.[[hc]] to *hourCycle*.
10. Let *localeData* be %DateTimeFormat%.[[LocaleData]].
11. Let *r* be ResolveLocale(%DateTimeFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %DateTimeFormat%.[[RelevantExtensionKeys]], *localeData*).
12. Set *dateTimeFormat*.[[Locale]] to *r*.[[locale]].
13. Set *dateTimeFormat*.[[Calendar]] to *r*.[[ca]].
14. Set *dateTimeFormat*.[[HourCycle]] to *r*.[[hc]].
15. Set *dateTimeFormat*.[[NumberingSystem]] to *r*.[[nu]].
16. Let *dataLocale* be *r*.[[dataLocale]].
17. Let *timeZone* be ? Get(*options*, **"timeZone"**).
18. If *timeZone* is not **undefined**, then
    a. Let *timeZone* be ? ToString(*timeZone*).
    b. If the result of IsValidTimeZoneName(*timeZone*) is **false**, then
        i. Throw a **RangeError** exception.
    c. Let *timeZone* be CanonicalizeTimeZoneName(*timeZone*).
19. Else,
    a. Let *timeZone* be DefaultTimeZone().
20. Set *dateTimeFormat*.[[TimeZone]] to *timeZone*.
21. Let *opt* be a new Record.
22. For each row of Table 5, except the header row, in table order, do
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *value* be ? GetOption(*options*, *prop*, **"string"**, « the strings given in the Values column of the row », **undefined**).
    c. Set *opt*.[[<*prop*>]] to *value*.
23. Let *dataLocaleData* be *localeData*.[[<*dataLocale*>]].
24. Let *formats* be *dataLocaleData*.[[formats]].
25. Let *matcher* be ? GetOption(*options*, **"formatMatcher"**, **"string"**, « **"basic"**, **"best fit"** », **"best fit"**).
26. If *matcher* is **"basic"**, then
    a. Let *bestFormat* be BasicFormatMatcher(*opt*, *formats*).
27. Else,
    a. Let *bestFormat* be BestFitFormatMatcher(*opt*, *formats*).
28. For each row in Table 5, except the header row, in table order, do
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *p* be *bestFormat*.[[<*prop*>]].
    c. If *p* not **undefined**, then
        i. Set *dateTimeFormat*'s internal slot whose name is the Internal Slot column of the row to *p*.
29. If *dateTimeFormat*.[[Hour]] is not **undefined**, then

a. Let *hcDefault* be *dataLocaleData*.[[hourCycle]].
b. Let *hc* be *dateTimeFormat*.[[HourCycle]].
c. If *hc* is **null**, then
    i. Set *hc* to *hcDefault*.
d. If *hour12* is not **undefined**, then
    i. If *hour12* is **true**, then
        1. If *hcDefault* is **"h11"** or **"h23"**, then
            a. Set *hc* to **"h11"**.
        2. Else,
            a. Set *hc* to **"h12"**.
    ii. Else,
        1. Assert: *hour12* is **false**.
        2. If *hcDefault* is **"h11"** or **"h23"**, then
            a. Set *hc* to **"h23"**.
        3. Else,
            a. Set *hc* to **"h24"**.
e. Set *dateTimeFormat*.[[HourCycle]] to *hc*.
f. If *dateTimeformat*.[[HourCycle]] is **"h11"** or **"h12"**, then
    i. Let *pattern* be *bestFormat*.[[pattern12]].
g. Else,
    i. Let *pattern* be *bestFormat*.[[pattern]].

30. Else,
    a. Set *dateTimeFormat*.[[HourCycle]] to **undefined**.
    b. Let *pattern* be *bestFormat*.[[pattern]].
31. Set *dateTimeFormat*.[[Pattern]] to *pattern*.
32. Return *dateTimeFormat*.

12.1.2 ToDateTimeOptions (*options*, *required*, *defaults*)

When the ToDateTimeOptions abstract operation is called with arguments *options*, *required*, and *defaults*, the following steps are taken:

1. If *options* is **undefined**, let *options* be **null**; otherwise let *options* be ? ToObject(*options*).
2. Let *options* be ObjectCreate(*options*).
3. Let *needDefaults* be **true**.
4. If *required* is **"date"** or **"any"**, then
    a. For each of the property names **"weekday"**, **"year"**, **"month"**, **"day"**, do
        i. Let *prop* be the property name.
        ii. Let *value* be ? Get(*options*, *prop*).
        iii. If *value* is not **undefined**, let *needDefaults* be **false**.
5. If *required* is **"time"** or **"any"**, then
    a. For each of the property names **"hour"**, **"minute"**, **"second"**, do
        i. Let *prop* be the property name.
        ii. Let *value* be ? Get(*options*, *prop*).
        iii. If *value* is not **undefined**, let *needDefaults* be **false**.
6. If *needDefaults* is **true** and *defaults* is either **"date"** or **"all"**, then
    a. For each of the property names **"year"**, **"month"**, **"day"**, do
        i. Perform ? CreateDataPropertyOrThrow(*options*, *prop*, **"numeric"**).

7. If *needDefaults* is **true** and *defaults* is either `"time"` or `"all"`, then
    a. For each of the property names `"hour"`, `"minute"`, `"second"`, do
        i. Perform ? CreateDataPropertyOrThrow(*options*, *prop*, `"numeric"`).
8. Return *options*.

12.1.3 BasicFormatMatcher (*options*, *formats*)

When the BasicFormatMatcher abstract operation is called with two arguments *options* and *formats*, the following steps are taken:

1. Let *removalPenalty* be 120.
2. Let *additionPenalty* be 20.
3. Let *longLessPenalty* be 8.
4. Let *longMorePenalty* be 6.
5. Let *shortLessPenalty* be 6.
6. Let *shortMorePenalty* be 3.
7. Let *bestScore* be **-Infinity**.
8. Let *bestFormat* be **undefined**.
9. Assert: Type(*formats*) is List.
10. For each element *format* of *formats* in List order, do
    a. Let *score* be 0.
    b. For each *property* shown in Table 5, do
        i. Let *optionsProp* be *options*.[[<*property*>]].
        ii. Let *formatProp* be *format*.[[<*property*>]].
        iii. If *optionsProp* is **undefined** and *formatProp* is not **undefined**, then decrease *score* by *additionPenalty*.
        iv. Else if *optionsProp* is not **undefined** and *formatProp* is **undefined**, then decrease *score* by *removalPenalty*.
        v. Else if *optionsProp* ≠ *formatProp*,
            1. Let *values* be « `"2-digit"`, `"numeric"`, `"narrow"`, `"short"`, `"long"` ».
            2. Let *optionsPropIndex* be the index of *optionsProp* within *values*.
            3. Let *formatPropIndex* be the index of *formatProp* within *values*.
            4. Let *delta* be max(min(*formatPropIndex* - *optionsPropIndex*, 2), -2).
            5. If *delta* = 2, decrease *score* by *longMorePenalty*.
            6. Else if *delta* = 1, decrease *score* by *shortMorePenalty*.
            7. Else if *delta* = -1, decrease *score* by *shortLessPenalty*.
            8. Else if *delta* = -2, decrease *score* by *longLessPenalty*.
    c. If *score* > *bestScore*, then
        i. Let *bestScore* be *score*.
        ii. Let *bestFormat* be *format*.
11. Return *bestFormat*.

12.1.4 BestFitFormatMatcher (*options*, *formats*)

When the BestFitFormatMatcher abstract operation is called with two arguments *options* and *formats*, it performs implementation dependent steps, which should return a set of component representations that a typical user of the selected locale would perceive as at least as good as the one returned by BasicFormatMatcher.

12.1.5 DateTime Format Functions

A DateTime format function is an anonymous built-in function that has a [[DateTimeFormat]] internal slot.

When a DateTime format function *F* is called with optional argument *date*, the following steps are taken:

1. Let *dtf* be *F*.[[DateTimeFormat]].
2. Assert: Type(*dtf*) is Object and *dtf* has an [[InitializedDateTimeFormat]] internal slot.
3. If *date* is not provided or is **undefined**, then
    a. Let *x* be Call(%Date_now%, **undefined**).
4. Else,
    a. Let *x* be ? ToNumber(*date*).
5. Return FormatDateTime(*dtf*, *x*).

The **`length`** property of a DateTime format function is 1.

12.1.6 PartitionDateTimePattern (*dateTimeFormat*, *x*)

The PartitionDateTimePattern abstract operation is called with arguments *dateTimeFormat* (which must be an object initialized as a DateTimeFormat) and *x* (which must be a Number value), interprets *x* as a time value as specified in ES2015, 20.3.1.1, and creates the corresponding parts according to the effective locale and the formatting options of *dateTimeFormat*. The following steps are taken:

1. Let *x* be TimeClip(*x*).
2. If *x* is **NaN**, throw a **RangeError** exception.
3. Let *locale* be *dateTimeFormat*.[[Locale]].
4. Let *nfOptions* be ObjectCreate(**null**).
5. Perform ! CreateDataPropertyOrThrow(*nfOptions*, **`"useGrouping"`**, **false**).
6. Let *nf* be ? Construct(%NumberFormat%, « *locale*, *nfOptions* »).
7. Let *nf2Options* be ObjectCreate(**null**).
8. Perform ! CreateDataPropertyOrThrow(*nf2Options*, **`"minimumIntegerDigits"`**, 2).
9. Perform ! CreateDataPropertyOrThrow(*nf2Options*, **`"useGrouping"`**, **false**).
10. Let *nf2* be ? Construct(%NumberFormat%, « *locale*, *nf2Options* »).
11. Let *tm* be ToLocalTime(*x*, *dateTimeFormat*.[[Calendar]], *dateTimeFormat*.[[TimeZone]]).
12. Let *pattern* be *dateTimeFormat*.[[Pattern]].
13. Let *result* be a new empty List.
14. Let *beginIndex* be Call(%StringProto_indexOf%, *pattern*, « **`"{"`**, 0 »).
15. Let *endIndex* be 0.
16. Let *nextIndex* be 0.
17. Let *length* be the number of code units in *pattern*.
18. Repeat, while *beginIndex* is an integer index into *pattern*
    a. Set *endIndex* to Call(%StringProto_indexOf%, *pattern*, « **`"}"`**, *beginIndex* »).
    b. Assert: *endIndex* is greater than *beginIndex*.
    c. If *beginIndex* is greater than *nextIndex*, then
        i. Let *literal* be a substring of *pattern* from position *nextIndex*, inclusive, to position *beginIndex*, exclusive.
        ii. Add new part record { [[Type]]: **`"literal"`**, [[Value]]: *literal* } as a new element of the list *result*.
    d. Let *p* be the substring of *pattern* from position *beginIndex*, exclusive, to position *endIndex*, exclusive.
    e. If *p* matches a Property column of the row in Table 5, then
        i. Let *f* be the value of *dateTimeFormat*'s internal slot whose name is the Internal Slot column of the

matching row.

ii. Let *v* be the value of *tm*'s field whose name is the Internal Slot column of the matching row.

iii. If *p* is **"year"** and *v* ≤ 0, let *v* be 1 - *v*.

iv. If *p* is **"month"**, increase *v* by 1.

v. If *p* is **"hour"** and *dateTimeFormat*.[[HourCycle]] is **"h11"** or **"h12"**, then

1. Let *v* be *v* modulo 12.
2. If *v* is 0 and *dateTimeFormat*.[[HourCycle]] is **"h12"**, let *v* be 12.

vi. If *p* is **"hour"** and *dateTimeFormat*.[[HourCycle]] is **"h24"**, then

1. If *v* is 0, let *v* be 24.

vii. If *f* is **"numeric"**, then

1. Let *fv* be FormatNumber(*nf*, *v*).

viii. Else if *f* is **"2-digit"**, then

1. Let *fv* be FormatNumber(*nf2*, *v*).
2. If the **length** property of *fv* is greater than 2, let *fv* be the substring of *fv* containing the last two characters.

ix. Else if *f* is **"narrow"**, **"short"**, or **"long"**, then let *fv* be a String value representing *f* in the desired form; the String value depends upon the implementation and the effective locale and calendar of *dateTimeFormat*. If *p* is **"month"**, then the String value may also depend on whether *dateTimeFormat* has a [[Day]] internal slot. If *p* is **"timeZoneName"**, then the String value may also depend on the value of the [[inDST]] field of *tm*. If *p* is **"era"**, then the String value may also depend on whether *dateTimeFormat* has a [[Era]] internal slot and if the implementation does not have a localized representation of *f*, then use *f* itself.

x. Add new part record { [[Type]]: *p*, [[Value]]: *fv* } as a new element of the list *result*.

f. Else if *p* is equal to **"ampm"**, then

i. Let *v* be *tm*.[[hour]].

ii. If *v* is greater than 11, then

1. Let *fv* be an implementation and locale dependent String value representing **"post meridiem"**.

iii. Else,

1. Let *fv* be an implementation and locale dependent String value representing **"ante meridiem"**.

iv. Add new part record { [[Type]]: **"dayPeriod"**, [[Value]]: *fv* } as a new element of the list *result*.

g. Else,

i. Let *unknown* be an implementation-, locale-, and numbering system-dependent String based on *x* and *p*.

ii. Append a new Record { [[Type]]: **"unknown"**, [[Value]]: *unknown* } as the last element of *result*.

h. Set *nextIndex* to *endIndex* + 1.

i. Set *beginIndex* to Call(%StringProto_indexOf%, *pattern*, « **"{"**, *nextIndex* »).

19. If *nextIndex* is less than *length*, then

a. Let *literal* be the substring of *pattern* from position *nextIndex*, exclusive, to position *length*, exclusive.

b. Add new part record { [[Type]]: **"literal"**, [[Value]]: *literal* } as a new element of the list *result*.

20. Return *result*.

NOTE 1

It is recommended that implementations use the locale and calendar dependent strings provided by the Common Locale Data Repository (available at http://cldr.unicode.org), and use CLDR **"abbreviated"** strings for DateTimeFormat **"short"** strings, and CLDR **"wide"** strings for DateTimeFormat **"long"** strings.

NOTE 2

It is recommended that implementations use the time zone information of the IANA Time Zone Database.

12.1.7 FormatDateTime(*dateTimeFormat*, *x*)

The FormatDateTime abstract operation is called with arguments *dateTimeFormat* (which must be an object initialized as a DateTimeFormat) and *x* (which must be a Number value), and performs the following steps:

1. Let *parts* be ? PartitionDateTimePattern(*dateTimeFormat*, *x*).
2. Let *result* be the empty String.
3. For each *part* in *parts*, do
   a. Set *result* to a String value produced by concatenating *result* and *part*.[[Value]].
4. Return *result*.

12.1.8 FormatDateTimeToParts (*dateTimeFormat*, *x*)

The FormatDateTimeToParts abstract operation is called with arguments *dateTimeFormat* (which must be an object initialized as a DateTimeFormat) and *x* (which must be a Number value), and performs the following steps:

1. Let *parts* be ? PartitionDateTimePattern(*dateTimeFormat*, *x*).
2. Let *result* be ArrayCreate(0).
3. Let *n* be 0.
4. For each *part* in *parts*, do
   a. Let *O* be ObjectCreate(%ObjectPrototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, `"type"`, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, `"value"`, *part*.[[Value]]).
   d. Perform ! CreateDataProperty(*result*, ! ToString(*n*), *O*).
   e. Increment *n* by 1.
5. Return *result*.

12.1.9 ToLocalTime (*date*, *calendar*, *timeZone*)

When the ToLocalTime abstract operation is called with arguments *date*, *calendar*, and *timeZone*, the following steps are taken:

1. Apply calendrical calculations on *date* for the given *calendar* and *timeZone* to produce weekday, era, year, month, day, hour, minute, second, and inDST values. The calculations should use best available information about the specified *calendar* and *timeZone*, including current and historical information about time zone offsets from UTC and daylight saving time rules. If the *calendar* is `"gregory"`, then the calculations must match the algorithms specified in ES2020, 20.3.1.
2. Return a Record with fields [[weekday]], [[era]], [[year]], [[month]], [[day]], [[hour]], [[minute]], [[second]], and [[inDST]], each with the corresponding calculated value.

NOTE
It is recommended that implementations use the time zone information of the IANA Time Zone Database.

12.1.10 UnwrapDateTimeFormat(*dtf*)

The UnwrapDateTimeFormat abstract operation gets the underlying DateTimeFormat operation for various methods which implement ECMA-402 v1 semantics for supporting initializing existing Intl objects.

1. Assert: Type(*dtf*) is Object.

2. If *dtf* does not have an [[InitializedDateTimeFormat]] internal slot and ? InstanceofOperator(*dtf*, %DateTimeFormat%) is **true**, then
    a. Let *dtf* be ? Get(*dtf*, %Intl%.[[FallbackSymbol]]).

2. If Type(*dtf*) is not Object or *dtf* does not have an [[InitializedDateTimeFormat]] internal slot, then
    a. Throw a **TypeError** exception.
3. Return *dtf*.

NOTE
See 8.1 Note 1 for the motivation of the normative optional text.

12.2 The Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor is the %DateTimeFormat% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

12.2.1 Intl.DateTimeFormat ([*locales* [, *options*]])

When the `Intl.DateTimeFormat` function is called with optional arguments *locales* and *options*, the following steps are taken:

1. If NewTarget is **undefined**, let *newTarget* be the active function object, else let *newTarget* be NewTarget.
2. Let *dateTimeFormat* be ? OrdinaryCreateFromConstructor(*newTarget*, `"%DateTimeFormatPrototype%"`, « [[InitializedDateTimeFormat]], [[Locale]], [[Calendar]], [[NumberingSystem]], [[TimeZone]], [[Weekday]], [[Era]], [[Year]], [[Month]], [[Day]], [[Hour]], [[Minute]], [[Second]], [[TimeZoneName]], [[HourCycle]], [[Pattern]], [[BoundFormat]] »).
3. Perform ? InitializeDateTimeFormat(*dateTimeFormat*, *locales*, *options*).

4. Let *this* be the **this** value.
5. If NewTarget is **undefined** and ? InstanceofOperator(*this*, %DateTimeFormat%) is **true**, then
    a. Perform ? DefinePropertyOrThrow(*this*, %Intl%.[[FallbackSymbol]], PropertyDescriptor{ [[Value]]: *dateTimeFormat*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
    b. Return *this*.

6. Return *dateTimeFormat*.

NOTE
See 8.1 Note 1 for the motivation of the normative optional text.

12.3 Properties of the Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor has the following properties:

12.3.1 Intl.DateTimeFormat.prototype

The value of `Intl.DateTimeFormat.prototype` is %DateTimeFormatPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

12.3.2 Intl.DateTimeFormat.supportedLocalesOf (*locales* [, *options*])

When the `supportedLocalesOf` method is called with arguments *locales* and *options*, the following steps are taken:

1. Let *availableLocales* be %DateTimeFormat%.[[AvailableLocales]].
2. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
3. Return ? SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the `length` property of the **supportedLocalesOf** method is 1.

12.3.3 Internal slots

The value of the [[AvailableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[RelevantExtensionKeys]] internal slot is « `"ca"`, `"nu"`, `"hc"` ».

NOTE 1
Unicode Technical Standard 35 describes three locale extension keys that are relevant to date and time formatting, `"ca"` for calendar, `"tz"` for time zone, `"hc"` for hour cycle, and implicitly `"nu"` for the numbering system of the number format used for numbers within the date format. DateTimeFormat, however, requires that the time zone is specified through the timeZone property in the options objects.

The value of the [[LocaleData]] internal slot is implementation defined within the constraints described in 9.1 and the following additional constraints:

The list that is the value of the `"nu"` field of any locale field of [[LocaleData]] must not include the values `"native"`, `"traditio"`, or `"finance"`.
[[LocaleData]].[[<*locale*>]].[[hc]] must be « **null**, `"h11"`, `"h12"`, `"h23"`, `"h24"` » for all locale values *locale*.
[[LocaleData]].[[<*locale*>]] must have an [[hourCycle]] field with a String value equal to `"h11"`, `"h12"`, `"h23"`, or `"h24"` for all locale values *locale*.
[[LocaleData]][[locale] must have a formats field for all locale values. The value of this field must be a list of records, each of which has a subset of the fields shown in Table 5, where each field must have one of the values specified for the field in Table 5. Multiple records in a list may use the same subset of the fields as long as they have different values for the fields. The following subsets must be available for each locale:
weekday, year, month, day, hour, minute, second
weekday, year, month, day
year, month, day
year, month
month, day
hour, minute, second
hour, minute
Each of the records must also have a pattern field, whose value is a String value that contains for each of the date and time format component fields of the record a substring starting with `"{"`, followed by the name of the field, followed by `"}"`. If the record has an hour field, it must also have a pattern12 field, whose value is a String value that, in addition to the substrings of the pattern field, contains a substring `"{ampm}"`.

EXAMPLE An implementation might include the following record as part of its English locale data: {[[hour]]:

"numeric", [[minute]]: **"2-digit"**, [[second]]: **"2-digit"**, [[pattern]]: **"{hour}:{minute}:{second}"**, [[pattern12]]: **"{hour}:{minute}:{second} {ampm}"**}.

NOTE 2

It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at http://cldr.unicode.org).

12.4 Properties of the Intl.DateTimeFormat Prototype Object

The Intl.DateTimeFormat prototype object is itself an ordinary object. %DateTimeFormatPrototype% is not an Intl.DateTimeFormat instance and does not have an [[InitializedDateTimeFormat]] internal slot or any of the other internal slots of Intl.DateTimeFormat instance objects.

12.4.1 Intl.DateTimeFormat.prototype.constructor

The initial value of **`Intl.DateTimeFormat.prototype.constructor`** is the intrinsic object %DateTimeFormat%.

12.4.2 Intl.DateTimeFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value **"Object"**.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

12.4.3 get Intl.DateTimeFormat.prototype.format

Intl.DateTimeFormat.prototype.format is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *dtf* be **this** value.
2. If Type(*dtf*) is not Object, throw a **TypeError** exception.
3. Let *dtf* be ? UnwrapDateTimeFormat(*dtf*).
4. If *dtf*.[[BoundFormat]] is **undefined**, then
   a. Let *F* be a new built-in function object as defined in DateTime Format Functions (12.1.5).
   b. Set *F*.[[DateTimeFormat]] to *dtf*.
   c. Set *dtf*.[[BoundFormat]] to *F*.
5. Return *dtf*.[[BoundFormat]].

NOTE

The returned function is bound to *dtf* so that it can be passed directly to **`Array.prototype.map`** or other functions. This is considered a historical artefact, as part of a convention which is no longer followed for new features, but is preserved to maintain compatibility with existing programs.

12.4.4 Intl.DateTimeFormat.prototype.formatToParts (*date*)

When the **`formatToParts`** method is called with an argument *date*, the following steps are taken:

1. Let *dtf* be **this** value.
2. If Type(*dtf*) is not Object, throw a **TypeError** exception.
3. If *dtf* does not have an [[InitializedDateTimeFormat]] internal slot, throw a **TypeError** exception.
4. If *date* is **undefined**, then
   a. Let *x* be Call(%Date_now%, **undefined**).
5. Else,
   a. Let *x* be ? ToNumber(*date*).
6. Return ? FormatDateTimeToParts(*dtf*, *x*).

12.4.5 Intl.DateTimeFormat.prototype.resolvedOptions ()

This function provides access to the locale and formatting options computed during initialization of the object.

1. Let *dtf* be **this** value.
2. If Type(*dtf*) is not Object, throw a **TypeError** exception.
3. Let *dtf* be ? UnwrapDateTimeFormat(*dtf*).
4. Let *options* be ! ObjectCreate(%ObjectPrototype%).
5. For each row of Table 6, except the header row, in table order, do
   a. Let *p* be the Property value of the current row.
   b. If *p* is **`"hour12"`**, then
      i. Let *hc* be *dtf*.[[HourCycle]].
      ii. If *hc* is **`"h11"`** or **`"h12"`**, let *v* be **true**.
      iii. Else if, *hc* is **`"h23"`** or **`"h24"`**, let *v* be **false**.
      iv. Else, let *v* be **undefined**.
   c. Else,
      i. Let *v* be the value of *dtf*'s internal slot whose name is the Internal Slot value of the current row.
   d. If *v* is not **undefined**, then
      i. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 6: Resolved Options of DateTimeFormat Instances

Internal Slot	Property
[[Locale]]	`"locale"`
[[Calendar]]	`"calendar"`
[[NumberingSystem]]	`"numberingSystem"`
[[TimeZone]]	`"timeZone"`
[[HourCycle]]	`"hourCycle"`
	`"hour12"`
[[Weekday]]	`"weekday"`
[[Era]]	`"era"`
[[Year]]	`"year"`
[[Month]]	`"month"`

Internal Slot	Property
[[Day]]	**`"day"`**
[[Hour]]	**`"hour"`**
[[Minute]]	**`"minute"`**
[[Second]]	**`"second"`**
[[TimeZoneName]]	**`"timeZoneName"`**

For web compatibility reasons, if the property hourCycle is set, the hour12 property should be set to **true** when hourCycle is **`"h11"`** or **`"h12"`**, or to **false** when hourCycle is **`"h23"`** or **`"h24"`**.

NOTE 1
In this version of the ECMAScript 2020 Internationalization API, the timeZone property will be the name of the default time zone if no timeZone property was provided in the options object provided to the Intl.DateTimeFormat constructor. The first edition left the timeZone property **undefined** in this case.

NOTE 2
For compatibility with versions prior to the fifth edition, the **`"hour12"`** property is set in addition to the **`"hourCycle"`** property.

12.5 Properties of Intl.DateTimeFormat Instances

Intl.DateTimeFormat instances inherit properties from %DateTimeFormatPrototype%.

Intl.DateTimeFormat instances have an [[InitializedDateTimeFormat]] internal slot.

Intl.DateTimeFormat instances also have several internal slots that are computed by the constructor:

- [[Locale]] is a String value with the language tag of the locale whose localization is used for formatting.
- [[Calendar]] is a String value with the **`"type"`** given in Unicode Technical Standard 35 for the calendar used for formatting.
- [[NumberingSystem]] is a String value with the **`"type"`** given in Unicode Technical Standard 35 for the numbering system used for formatting.
- [[TimeZone]] is a String value with the IANA time zone name of the time zone used for formatting.
- [[Weekday]], [[Era]], [[Year]], [[Month]], [[Day]], [[Hour]], [[Minute]], [[Second]], [[TimeZoneName]] are each either **undefined**, indicating that the component is not used for formatting, or one of the String values given in Table 5, indicating how the component should be presented in the formatted output.
- [[HourCycle]] is a String value indicating whether the 12-hour format (**`"h11"`**, **`"h12"`**) or the 24-hour format (**`"h23"`**, **`"h24"`**) should be used. **`"h11"`** and **`"h23"`** start with hour 0 and go up to 11 and 23 respectively. **`"h12"`** and **`"h24"`** start with hour 1 and go up to 12 and 24. [[HourCycle]] is only used when [[Hour]] is not **undefined**.
- [[Pattern]] is a String value as described in 12.3.3.

Finally, Intl.DateTimeFormat instances have a [[BoundFormat]] internal slot that caches the function returned by the format accessor (12.4.3).

13 PluralRules Objects

13.1 Abstract Operations for PluralRules Objects

13.1.1 InitializePluralRules (*pluralRules*, *locales*, *options*)

The abstract operation InitializePluralRules accepts the arguments *pluralRules* (which must be an object), *locales*, and *options*. It initializes *pluralRules* as a PluralRules object. The following steps are taken:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. If *options* is **undefined**, then
    a. Let *options* be ObjectCreate(**null**).
3. Else
    a. Let *options* be ? ToObject(*options*).
4. Let *opt* be a new Record.
5. Let *matcher* be ? GetOption(*options*, `"localeMatcher"`, `"string"`, « `"lookup"`, `"best fit"` », `"best fit"`).
6. Set *opt*.[[localeMatcher]] to *matcher*.
7. Let *t* be ? GetOption(*options*, `"type"`, `"string"`, « `"cardinal"`, `"ordinal"` », `"cardinal"`).
8. Set *pluralRules*.[[Type]] to *t*.
9. Perform ? SetNumberFormatDigitOptions(*pluralRules*, *options*, **0**, **3**).
10. Let *localeData* be %PluralRules%.[[LocaleData]].
11. Let *r* be ResolveLocale(%PluralRules%.[[AvailableLocales]], *requestedLocales*, *opt*, %PluralRules%.[[RelevantExtensionKeys]], *localeData*).
12. Set *pluralRules*.[[Locale]] to the value of *r*.[[locale]].
13. Return *pluralRules*.

13.1.2 GetOperands (*s*)

When the GetOperands abstract operation is called with argument *s*, it performs the following steps:

1. Assert: Type(*s*) is String.
2. Let *n* be ! ToNumber(*s*).
3. Assert: *n* is finite.
4. Let *dp* be ! Call(%StringProto_indexOf%, *s*, « `"."` »).
5. If *dp* = -1, then
    a. Set *iv* to *n*.
    b. Let *f* be 0.
    c. Let *v* be 0.
6. Else,
    a. Let *iv* be the substring of *s* from position 0, inclusive, to position *dp*, exclusive.
    b. Let *fv* be the substring of *s* from position *dp*, exclusive, to the end of *s*.
    c. Let *f* be ! ToNumber(*fv*).
    d. Let *v* be the length of *fv*.
7. Let *i* be abs(! ToNumber(*iv*)).
8. If *f* ≠ 0, then
    a. Let *ft* be the value of *fv* stripped of trailing `"0"`.

    b. Let *w* be the length of *ft*.
    c. Let *t* be ! ToNumber(*ft*).
9. Else,
    a. Let *w* be 0.
    b. Let *t* be 0.
10. Return a new Record { [[Number]]: *n*, [[IntegerDigits]]: *i*, [[NumberOfFractionDigits]]: *v*, [[NumberOfFractionDigitsWithoutTrailing]]: *w*, [[FractionDigits]]: *f*, [[FractionDigitsWithoutTrailing]]: *t* }.

Table 7: Plural Rules Operands Record Fields

Internal Slot	Type	Description
[[Number]]	Number	Absolute value of the source number (integer and decimals)
[[IntegerDigits]]	Number	Number of digits of [[Number]].
[[NumberOfFractionDigits]]	Number	Number of visible fraction digits in [[Number]], *with* trailing zeros.
[[NumberOfFractionDigitsWithoutTrailing]]	Number	Number of visible fraction digits in [[Number]], *without* trailing zeros.
[[FractionDigits]]	Number	Number of visible fractional digits in [[Number]], *with* trailing zeros.
[[FractionDigitsWithoutTrailing]]	Number	Number of visible fractional digits in [[Number]], *without* trailing zeros.

13.1.3 PluralRuleSelect (*locale*, *type*, *n*, *operands*)

When the PluralRuleSelect abstract operation is called with four arguments, it performs an implementation-dependent algorithm to map *n* to the appropriate plural representation of the Plural Rules Operands Record *operands* by selecting the rules denoted by *type* for the corresponding *locale*, or the String value `"other"`.

13.1.4 ResolvePlural (*pluralRules*, *n*)

When the ResolvePlural abstract operation is called with arguments *pluralRules* (which must be an object initialized as a PluralRules) and *n* (which must be a Number value), it returns a String value representing the plural form of *n* according to the effective locale and the options of *pluralRules*. The following steps are taken:

1. Assert: Type(*pluralRules*) is Object.
2. Assert: *pluralRules* has an [[InitializedPluralRules]] internal slot.
3. Assert: Type(*n*) is Number.
4. If *n* is not a finite Number, then
    a. Return `"other"`.
5. Let *locale* be *pluralRules*.[[Locale]].
6. Let *type* be *pluralRules*.[[Type]].
7. Let *s* be ! FormatNumberToString(*pluralRules*, *n*).
8. Let *operands* be ? *GetOperands*(*s*).
9. Return ? PluralRuleSelect(*locale*, *type*, *n*, *operands*).

13.2 The Intl.PluralRules Constructor

The PluralRules constructor is the %PluralRules% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.

13.2.1 Intl.PluralRules ([*locales* [, *options*]])

When the `Intl.PluralRules` function is called with optional arguments *locales* and *options*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *pluralRules* be ? OrdinaryCreateFromConstructor(*newTarget*, "%PluralRulesPrototype%", « [[InitializedPluralRules]], [[Locale]], [[Type]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]] »).
3. Return ? InitializePluralRules(*pluralRules*, *locales*, *options*).

13.3 Properties of the Intl.PluralRules Constructor

The Intl.PluralRules constructor has the following properties:

13.3.1 Intl.PluralRules.prototype

The value of `Intl.PluralRules.prototype` is %PluralRulesPrototype%.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }.

13.3.2 Intl.PluralRules.supportedLocalesOf (*locales* [, *options*])

When the `supportedLocalesOf` method is called with arguments *locales* and *options*, the following steps are taken:

1. Let *availableLocales* be %PluralRules%.[[AvailableLocales]].
2. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
3. Return ? SupportedLocales(*availableLocales*, *requestedLocales*, *options*).

The value of the `length` property of the **supportedLocalesOf** method is 1.

13.3.3 Internal slots

The value of the [[AvailableLocales]] internal slot is implementation defined within the constraints described in 9.1.

The value of the [[RelevantExtensionKeys]] internal slot is **[]**.

NOTE 1
Unicode Technical Standard 35 describes no locale extension keys that are relevant to the pluralization process.

The value of the [[LocaleData]] internal slot is implementation defined within the constraints described in 9.1.

NOTE 2
It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available

at http://cldr.unicode.org).

13.4 Properties of the Intl.PluralRules Prototype Object

The Intl.PluralRules prototype object is itself an ordinary object. %PluralRulesPrototype% is not an Intl.PluralRules instance and does not have an [[InitializedPluralRules]] internal slot or any of the other internal slots of Intl.PluralRules instance objects.

13.4.1 Intl.PluralRules.prototype.constructor

The initial value of `Intl.PluralRules.prototype.constructor` is the intrinsic object %PluralRules%.

13.4.2 Intl.PluralRules.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the string value `"Object"`.

This property has the attributes { [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **true** }.

13.4.3 Intl.PluralRules.prototype.select(*value*)

When the `select` method is called with an argument *value*, the following steps are taken:

1. Let *pr* be the **this** value.
2. If Type(*pr*) is not Object, throw a **TypeError** exception.
3. If *pr* does not have an [[InitializedPluralRules]] internal slot, throw a **TypeError** exception.
4. Let *n* be ? ToNumber(*value*).
5. Return ? ResolvePlural(*pr*, *n*).

13.4.4 Intl.PluralRules.prototype.resolvedOptions ()

This function provides access to the locale and options computed during initialization of the object.

1. Let *pr* be the **this** value.
2. If Type(*pr*) is not Object, throw a **TypeError** exception.
3. If *pr* does not have an [[InitializedPluralRules]] internal slot, throw a **TypeError** exception.
4. Let *options* be ! ObjectCreate(%ObjectPrototype%).
5. For each row of Table 8, except the header row, in table order, do
   a. Let *p* be the Property value of the current row.
   b. Let *v* be the value of *pr*'s internal slot whose name is the Internal Slot value of the current row.
   c. If *v* is not **undefined**, then
      i. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Let *pluralCategories* be a List of Strings representing the possible results of PluralRuleSelect for the selected locale *pr*.[[Locale]]. This List consists of unique string values, from the the list `"zero"`, `"one"`, `"two"`, `"few"`, `"many"` and `"other"`, that are relevant for the locale whose localization is specified in LDML Language Plural Rules.
7. Perform ! CreateDataProperty(*options*, `"pluralCategories"`, CreateArrayFromList(*pluralCategories*)).

8. Return *options*.

Table 8: Resolved Options of PluralRules Instances

Internal Slot	Property
[[Locale]]	**`"locale"`**
[[Type]]	**`"type"`**
[[MinimumIntegerDigits]]	**`"minimumIntegerDigits"`**
[[MinimumFractionDigits]]	**`"minimumFractionDigits"`**
[[MaximumFractionDigits]]	**`"maximumFractionDigits"`**
[[MinimumSignificantDigits]]	**`"minimumSignificantDigits"`**
[[MaximumSignificantDigits]]	**`"maximumSignificantDigits"`**

13.5 Properties of Intl.PluralRules Instances

Intl.PluralRules instances inherit properties from %PluralRulesPrototype%.

Intl.PluralRules instances have an [[InitializedPluralRules]] internal slots.

Intl.PluralRules instances also have several internal slots that are computed by the constructor:

[[Locale]] is a String value with the language tag of the locale whose localization is used by the plural rules.
[[Type]] is one of the String values **`"cardinal"`** or **`"ordinal"`**, identifying the plural rules used.
[[MinimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used.
[[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.
[[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be used. Either none or both of these properties are present; if they are, they override minimum and maximum integer and fraction digits.

14 Locale Sensitive Functions of the ECMAScript Language Specification

The ECMAScript Language Specification, edition 10 or successor, describes several locale sensitive functions. An ECMAScript implementation that implements this Internationalization API Specification shall implement these functions as described here.

NOTE
The Collator, NumberFormat, or DateTimeFormat objects created in the algorithms in this clause are only used within these algorithms. They are never directly accessed by ECMAScript code and need not actually exist within an implementation.

14.1 Properties of the String Prototype Object

14.1.1 String.prototype.localeCompare (*that* [, *locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 21.1.3.10.

When the `localeCompare` method is called with argument *that* and optional arguments *locales*, and *options*, the following steps are taken:

1. Let *O* be RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *thatValue* be ? ToString(*that*).
4. Let *collator* be ? Construct(%Collator%, « *locales*, *options* »).
5. Return CompareStrings(*collator*, *S*, *thatValue*).

The value of the `length` property of the **localeCompare** method is 1.

NOTE 1
The `localeCompare` method itself is not directly suitable as an argument to **Array.prototype.sort** because the latter requires a function of two arguments.

NOTE 2
The `localeCompare` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

14.1.2 String.prototype.toLocaleLowerCase ([*locales*])

This definition supersedes the definition provided in ES2020, 21.1.3.22.

This function interprets a string value as a sequence of code points, as described in ES2020, 6.1.4. The following steps are taken:

1. Let *O* be RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
4. If *requestedLocales* is not an empty List, then
   a. Let *requestedLocale* be *requestedLocales*[0].
5. Else,
   a. Let *requestedLocale* be DefaultLocale().
6. Let *noExtensionsLocale* be the String value that is *requestedLocale* with all Unicode locale extension sequences (6.2.1) removed.
7. Let *availableLocales* be a List with language tags that includes the languages for which the Unicode Character Database contains language sensitive case mappings. Implementations may add additional language tags if they support case mapping for additional locales.
8. Let *locale* be BestAvailableLocale(*availableLocales*, *noExtensionsLocale*).
9. If *locale* is **undefined**, let *locale* be `"und"`.
10. Let *cpList* be a List containing in order the code points of *S* as defined in ES2020, 6.1.4, starting at the first element of *S*.
11. Let *cuList* be a List where the elements are the result of a lower case transformation the ordered code points in *cpList* according to the Unicode Default Case Conversion algorithm or an implementation defined conversion

algorithm. A conforming implementation's lower case transformation algorithm must always yield the same *cpList* given the same *cuList* and locale.

12. Let *L* be a String whose elements are the UTF-16 Encoding (defined in ES2020, 6.1.4) of the code points of *cuList*.
13. Return *L*.

Lower case code point mappings may be derived according to a tailored version of the Default Case Conversion Algorithms of the Unicode Standard. Implementations may use locale specific tailoring defined in SpecialCasings.txt and/or CLDR and/or any other custom tailoring.

NOTE 1

The case mapping of some code points may produce multiple code points. In this case the result String may not be the same length as the source String. Because both `toLocaleUpperCase` and `toLocaleLowerCase` have context-sensitive behaviour, the functions are not symmetrical. In other words, `s.toLocaleUpperCase().toLocaleLowerCase()` is not necessarily equal to `s.toLocaleLowerCase()`.

NOTE 2

The `toLocaleLowerCase` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

14.1.3 String.prototype.toLocaleUpperCase ([*locales*])

This definition supersedes the definition provided in ES2020, 21.1.3.23.

This function interprets a string value as a sequence of code points, as described in ES2020, 6.1.4. This function behaves in exactly the same way as `String.prototype.toLocaleLowerCase`, except that characters are mapped to their *uppercase* equivalents. A conforming implementation's upper case transformation algorithm must always yield the same result given the same sequence of code points and locale.

NOTE

The `toLocaleUpperCase` function is intentionally generic; it does not require that its **this** value be a String object. Therefore, it can be transferred to other kinds of objects for use as a method.

14.2 Properties of the Number Prototype Object

The following definition(s) refer to the abstract operation thisNumberValue as defined in ES2020, 20.1.3.

14.2.1 Number.prototype.toLocaleString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 20.1.3.4.

When the `toLocaleString` method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be ? thisNumberValue(**this** value).
2. Let *numberFormat* be ? Construct(%NumberFormat%, « *locales*, *options* »).
3. Return FormatNumeric(*numberFormat*, *x*).

14.3 Properties of the BigInt Prototype Object

The following definition(s) refer to the abstract operation thisBigIntValue as defined in ES2019, .

14.3.1 BigInt.prototype.toLocaleString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2019, .

When the `toLocaleString` method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be ? thisBigIntValue(**this** value).
2. Let *numberFormat* be ? Construct(%NumberFormat%, « *locales*, *options* »).
3. Return FormatNumeric(*numberFormat*, *x*).

14.4 Properties of the Date Prototype Object

The following definition(s) refer to the abstract operation thisTimeValue as defined in ES2020, 20.3.4.

14.4.1 Date.prototype.toLocaleString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 20.3.4.39.

When the `toLocaleString` method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be ? thisTimeValue(**this** value).
2. If *x* is **NaN**, return `"Invalid Date"`.
3. Let *options* be ? ToDateTimeOptions(*options*, `"any"`, `"all"`).
4. Let *dateFormat* be ? Construct(%DateTimeFormat%, « *locales*, *options* »).
5. Return FormatDateTime(*dateFormat*, *x*).

14.4.2 Date.prototype.toLocaleDateString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 20.3.4.38.

When the `toLocaleDateString` method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be ? thisTimeValue(**this** value).
2. If *x* is **NaN**, return `"Invalid Date"`.
3. Let *options* be ? ToDateTimeOptions(*options*, `"date"`, `"date"`).
4. Let *dateFormat* be ? Construct(%DateTimeFormat%, « *locales*, *options* »).
5. Return FormatDateTime(*dateFormat*, *x*).

14.4.3 Date.prototype.toLocaleTimeString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 20.3.4.40.

When the **`toLocaleTimeString`** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *x* be ? thisTimeValue(**this** value).
2. If *x* is **NaN**, return **`"Invalid Date"`**.
3. Let *options* be ? ToDateTimeOptions(*options*, **`"time"`**, **`"time"`**).
4. Let *timeFormat* be ? Construct(%DateTimeFormat%, « *locales*, *options* »).
5. Return FormatDateTime(*timeFormat*, *x*).

14.5 Properties of the Array Prototype Object

14.5.1 Array.prototype.toLocaleString ([*locales* [, *options*]])

This definition supersedes the definition provided in ES2020, 22.1.3.27.

When the **`toLocaleString`** method is called with optional arguments *locales* and *options*, the following steps are taken:

1. Let *array* be ? ToObject(**this** value).
2. Let *len* be ? ToLength(? Get(*array*, **`"length"`**)).
3. Let *separator* be the String value for the list-separator String appropriate for the host environment's current locale (this is derived in an implementation-defined way).
4. Let *R* be the empty String.
5. Let *k* be 0.
6. Repeat, while *k* < *len*
    a. If *k* > 0, then
        i. Set *R* to the string-concatenation of *R* and *separator*.
    b. Let *nextElement* be ? Get(*array*, ! ToString(*k*)).
    c. If *nextElement* is not **undefined** or **null**, then
        i. Let *S* be ? ToString(? Invoke(*nextElement*, **`"toLocaleString"`**, « *locales*, *options* »)).
        ii. Set *R* to the string-concatenation of *R* and *S*.
    d. Increase *k* by 1.
7. Return *R*.

NOTE 1
This algorithm's steps mirror the steps taken in 22.1.3.27, with the exception that Invoke(*nextElement*, **`"toLocaleString"`**) now takes *locales* and *options* as arguments.

NOTE 2
The elements of the array are converted to Strings using their **toLocaleString** methods, and these Strings are then concatenated, separated by occurrences of a separator String that has been derived in an implementationdefined locale-specific way. The result of calling this function is intended to be analogous to the result of **toString**, except that the result of this function is intended to be locale-specific.

NOTE 3
The **`toLocaleString`** function is intentionally generic; it does not require that its **this** value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.

A Implementation Dependent Behaviour

The following aspects of the ECMAScript 2020 Internationalization API Specification are implementation dependent:

In all functionality:

- Additional values for some properties of *options* arguments (2)
- Canonicalization of extension subtag sequences beyond the rules of RFC 5646 (6.2.3)
- The default locale (6.2.4)
- The default time zone (6.4.3)
- The set of available locales for each constructor (9.1)
- The BestFitMatcher algorithm (9.2.4)
- The BestFitSupportedLocales algorithm (9.2.8)

In Collator:

- Support for the Unicode extensions keys kn, kf and the parallel options properties numeric, caseFirst (10.1.1)
- The set of supported `"co"` key values (collations) per locale beyond a default collation (10.2.3)
- The set of supported `"kn"` key values (numeric collation) per locale (10.2.3)
- The set of supported `"kf"` key values (case order) per locale (10.2.3)
- The default search sensitivity per locale (10.2.3)
- The sort order for each supported locale and options combination (10.3.3.1)

In NumberFormat:

- The set of supported `"nu"` key values (numbering systems) per locale (11.3.3)
- The patterns used for formatting positive and negative values as decimal, percent, or currency values per locale (11.1.7)
- Localized representations of **NaN** and **Infinity** (11.1.7)
- The implementation of numbering systems not listed in Table 3 (11.1.7)
- Localized decimal and grouping separators (11.1.7)
- Localized digit grouping schemata (11.1.7)
- Localized currency symbols and names (11.1.7)

In DateTimeFormat:

- The BestFitFormatMatcher algorithm (12.1.1)
- The set of supported `"ca"` key values (calendars) per locale (12.3.3)
- The set of supported `"nu"` key values (numbering systems) per locale (12.3.3)
- The default hourCycle setting per locale (12.3.3)
- The set of supported date-time formats per locale beyond a core set, including the representations used for each component and the associated patterns (12.3.3)
- Localized weekday names, era names, month names, am/pm indicators, and time zone names (12.1.7)
- The calendric calculations used for calendars other than `"gregory"`, and adjustments for local time zones and daylight saving time (12.1.7)

In PluralRules:

- List of Strings representing the possible results of plural selection and their corresponding order per locale. (13.1.1)

B Additions and Changes That Introduce Incompatibilities with Prior Editions

10.1, 11.2, 12.2 In ECMA-402, 1st Edition, constructors could be used to create Intl objects from arbitrary objects.

This is no longer possible in 2nd Edition.
12.4.3 In ECMA-402, 1st Edition, the *length* property of the function object *F* was set to 0. In 2nd Edition, *length* is set to 1.

C Colophon

This specification is authored on GitHub in a plaintext source format called Ecmarkup. Ecmarkup is an HTML and Markdown dialect that provides a framework and toolset for authoring ECMAScript specifications in plaintext and processing the specification into a full-featured HTML rendering that follows the editorial conventions for this document. Ecmarkup builds on and integrates a number of other formats and technologies including Grammarkdown for defining syntax and Ecmarkdown for authoring algorithm steps. PDF renderings of this specification are produced by printing the HTML rendering to a PDF.

Prior editions of this specification were authored using Word—the Ecmarkup source text that formed the basis of this edition was produced by converting the ECMAScript 2015 Word document to Ecmarkup using an automated conversion tool.

D Copyright & Software License

Ecma International

Rue du Rhone 114

CH-1204 Geneva

Tel: +41 22 849 6000

Fax: +41 22 849 6001

Web: https://ecma-international.org/

Copyright Notice