ECMA INTERNATIONAL

Standard ECMA-402

11th Edition / June 2024

ECMAScript® 2024 Internationalization API Specification

Standard

Rue du Rhône 114 CH-1204 Geneva T: +41 22 849 6000 F: +41 22 849 6001

Ecma International
Rue du Rhone 114 CH-1204 Geneva
Tel: +41 22 849 6000
Fax: +41 22 849 6001
Web: https://www.ecma-international.org

Contents

page

Introduction

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

The ECMAScript 2024 Internationalization API Specification (ECMA-402 11th Edition), provides key language sensitive functionality as a complement to the ECMAScript 2024 Language Specification (ECMA-262 15th 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* (https://unicode-org.github.io/icu-docs/), 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

Leo Balter, Valerie Young, Isaac Durazo
ECMA-402, 7th Edition Project Editors

Leo Balter, Richard Gibson
ECMA-402, 8th Edition Project Editors

Leo Balter, Richard Gibson, Ujjwal Sharma
ECMA-402, 9th Edition Project Editors

Richard Gibson, Ujjwal Sharma
ECMA-402, 10th & 11th Edition Project Editors

This Ecma Standard was developed by Technical Committee 39 and was adopted by the General Assembly of June 2024.

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 <https://github.com/tc39/ecma402/issues>, File a New Issue <https://github.com/tc39/ecma402/issues/new>
Pull Requests: All Pull Requests <https://github.com/tc39/ecma402/pulls>, Create a New Pull Request <https://github.com/tc39/ecma402/pulls/new>
Test Suite: Test262 <https://github.com/tc39/test262>
TC39-TG2:

- Convener: Shane F. Carr (@sffc <https://github.com/sffc>)
- Admin group: contact by email

Editors:

- Richard Gibson (@gibson042 <https://github.com/gibson042>)
- Ujjwal Sharma (@ryzokuken <https://github.com/ryzokuken>)

Community:

- Matrix: #tc39:matrix.org <https://matrix.to/#/#tc39:matrix.org>
- Matrix: #tc39-ecma402:matrix.org <https://matrix.to/#/#tc39-ecma402:matrix.org>

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

ECMAScript® 2024 Internationalization API Specification

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 this specification must conform to the ECMAScript 2024 Language Specification (ECMA-262 15th Edition, or successor), and must provide and support all the objects, properties, functions, and program semantics described in this specification. Nothing in this specification is intended to allow behaviour that is otherwise prohibited by ECMA-262, and any such conflict should be considered an editorial error rather than an override of constraints from ECMA-262.

A conforming implementation is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation is permitted to provide properties not described in this specification, and values for those properties, for objects that are described herein. 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"**, **"currencyDisplay"**, **"notation"**, **"compactDisplay"**, **"signDisplay"**, **"currencySign"**, and **"unitDisplay"** 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 7 in the DateTimeFormat constructor.
- The *options* property **"formatMatcher"** in the DateTimeFormat constructor.
- The *options* properties **"minimumIntegerDigits"**, **"minimumFractionDigits"**, **"maximumFractionDigits"**, and **"minimumSignificantDigits"** in the PluralRules constructor, provided that the additional values are interpreted as integer values higher than the specified limits.
- The *options* property **"type"** in the PluralRules constructor.
- The *options* property **"style"** and **"numeric"** in the RelativeTimeFormat constructor.
- The *options* property **"style"** and **"type"** in the DisplayNames 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 2024 Language Specification (ECMA-262 15th Edition, or successor).
https://www.ecma-international.org/publications/standards/Ecma-262.htm

> NOTE 1 Throughout this document, the phrase "es2024, *x*" (where x is a sequence of numbers separated by periods) may be used as shorthand for "ECMAScript 2024 Language Specification (ECMA-262 15th Edition, sub clause *x*)". Where *x* is followed by more such sequences of period-separated numbers, separated from each other by commas, each such sequence is also a shorthand for the corresponding sub clause of ECMA-262.

- 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 <https://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=64758>
- IETF RFC 4647, Matching of Language Tags, or successor <https://tools.ietf.org/html/rfc4647>
- IANA Time Zone Database <https://www.iana.org/time-zones/>
- The Unicode Standard <https://unicode.org/versions/latest>
- Unicode Standard Annex #29: Unicode Text Segmentation <https://unicode.org/reports/tr29/>
- Unicode Technical Standard #10: Unicode Collation Algorithm <https://unicode.org/reports/tr10/>
- Unicode Technical Standard #35: Unicode Locale Data Markup Language (LDML) <https://unicode.org/reports/tr35/>
  - Part 1 Core, Section 3 Unicode Language and Locale Identifiers <https://unicode.org/reports/tr35/#Unicode_Language_and_Locale_Identifiers>
  - Part 2 General, Section 6.2 Unit Identifiers <https://unicode.org/reports/tr35/tr35-general.html#Unit_Identifiers>
  - Part 3 Numbers, Section 5.1.1 Operands <https://unicode.org/reports/tr35/tr35-numbers.html#Operands>

NOTE 2 Sections of this specification that depend on these references are updated on a best-effort basis, but are not guaranteed to be up-to-date with those standards.

4 Overview

This section contains a non-normative overview of the ECMAScript 2024 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 world-wide 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 2024 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. This specification builds on that foundation 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

This specification is designed to complement the ECMAScript 2024 Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript 2024 Language Specification (ECMA-262 15th Edition, or successor) in whole or in part. This specification introduces new language values observable to ECMAScript code (such as the value of a [[FallbackSymbol]] internal slot and the set of values transitively reachable from %Intl% by property access), and also refines the definition of some functions specified in ECMA-262 (as described below). Neither category prohibits behaviour that is otherwise permitted for values and interfaces defined in ECMA-262, in order to support adoption of this specification by any implementation of ECMA-262.

This specification provides several key pieces of language-sensitive functionality that are required in most applications: String comparison (collation), number formatting, date and time formatting, relative time formatting, display names, list formatting, locale selection and operation, pluralization rules, case conversion, and text segmentation. While the ECMAScript 2024 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`**), their actual behaviour is left largely implemenation-defined. This 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 a service constructor to construct an object, specifying a list of preferred languages and options to configure its behaviour. The object provides a main function (**`compare`**, **`select`**, **`format`**, etc.), 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 2024 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 this specification in order to avoid name collisions.

NOTE While the API includes a variety of formatters, it does not provide any parsing facilities. This is intentional, has been discussed extensively, and concluded after weighing in all the benefits and drawbacks of including said functionality. See the discussion on the issue tracker <https://github.com/tc39/ecma402/issues/424>.

4.3 API Conventions

Every Intl constructor should behave as if defined by a class, throwing a **TypeError** exception when called as a function (without NewTarget). For backwards compatibility with past editions, this does not apply to %Intl.Collator%, %Intl.DateTimeFormat%, or %Intl.NumberFormat%, each of which construct and return a new object when called as a function.

NOTE In ECMA 402 v1, Intl constructors supported a mode of operation where calling them with an existing object as a receiver would add relevant internal slots to the receiver, effectively transforming it into an instance of the class. In ECMA 402 v2, this capability was removed, to avoid adding internal slots to 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 <https://github.com/tc39/ecma402/issues/57> for details.

4.4 Implementation Dependencies

Due to the nature of internationalization, this 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.

4.4.1 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 2024 Language Specification (ECMA-262 15th Edition), as es2024:

- Object Internal Methods and Internal Slots, as described in es2024, 6.1.7.2.
- Algorithm conventions, as described in es2024, 5.2, and the use of abstract operations, as described in es2024, 7.1, 7.2, 7.3, 7.4.
- Internal Slots, as described in es2024, 10.1.
- The List and Record Specification Type, as described in es2024, 6.2.2.

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 <https://tc39.es/ecma262/#sec-additional-ecmascript-features-for-web-browsers> 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 es2024, 6.1.7.4.

Table 1: Well-known Intrinsic Objects (Extensions)

Intrinsic Name	Global Name	ECMAScript Language Association
%Intl%	**Intl**	The **Intl** object (8)
%Intl.Collator%	**Intl.Collator**	The **Intl.Collator** constructor (10.1)
%Intl.DateTimeFormat%	**Intl.DateTimeFormat**	The **Intl.DateTimeFormat** constructor (11.1)

Table 1: Well-known Intrinsic Objects (Extensions) *(continued)*

Intrinsic Name	Global Name	ECMAScript Language Association
%Intl.DisplayNames%	`Intl.DisplayNames`	The `Intl.DisplayNames` constructor (12.1)
%Intl.ListFormat%	`Intl.ListFormat`	The `Intl.ListFormat` constructor (13.1)
%Intl.Locale%	`Intl.Locale`	The `Intl.Locale` constructor (14.1)
%Intl.NumberFormat%	`Intl.NumberFormat`	The `Intl.NumberFormat` constructor (15.1)
%Intl.PluralRules%	`Intl.PluralRules`	The `Intl.PluralRules` constructor (16.1)
%Intl.RelativeTimeFormat%	`Intl.RelativeTimeFormat`	The `Intl.RelativeTimeFormat` constructor (17.1)
%Intl.Segmenter%	`Intl.Segmenter`	The `Intl.Segmenter` constructor (18.1)
%IntlSegmentIteratorPrototype%		The prototype of Segment Iterator objects (18.6.2)
%IntlSegmentsPrototype%		The prototype of Segments objects (18.5.2)

6 Identification of Locales, Currencies, Time Zones, Measurement Units, Numbering Systems, Collations, and Calendars

This clause describes the String values used in this specification to identify locales, currencies, time zones, measurement units, numbering systems, collations, and calendars.

6.1 Case Sensitivity and Case Mapping

The String values used to identify locales, currencies, scripts, and time zones are interpreted in an ASCII-case-insensitive manner, treating the code units `0x0041` through `0x005A` (corresponding to Unicode characters LATIN CAPITAL LETTER A through LATIN CAPITAL LETTER Z) as equivalent to the corresponding code units `0x0061` through `0x007A` (corresponding to Unicode characters LATIN SMALL LETTER A through LATIN SMALL LETTER Z), both inclusive. No other case folding equivalences are applied.

NOTE For example, **"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`).

The *ASCII-uppercase* of a String value *S* is the String value derived from *S* by replacing each occurrence of an ASCII lowercase letter code unit (`0x0061` through `0x007A`, inclusive) with the corresponding ASCII uppercase letter code unit (`0x0041` through `0x005A`, inclusive) while preserving all other code units.

The *ASCII-lowercase* of a String value *S* is the String value derived from *S* by replacing each occurrence of an ASCII uppercase letter code unit (`0x0041` through `0x005A`, inclusive) with the corresponding ASCII lowercase letter code unit (`0x0061` through `0x007A`, inclusive) while preserving all other code units.

A String value *A* is an *ASCII-case-insensitive match* for String value *B* if the ASCII-uppercase of *A* is exactly the same sequence of code units as the ASCII-uppercase of *B*. A sequence of Unicode code points *A* is an ASCII-case-insensitive match for *B* if *B* is an ASCII-case-insensitive match for ! CodePointsToString(*A*).

6.2 Language Tags

This specification identifies locales using *Unicode BCP 47 locale identifiers* as defined by Unicode Technical Standard #35 Part 1 Core, Section 3.3 BCP 47 Conformance <https://unicode.org/reports/tr35/#BCP_47_Conformance>, and its algorithms refer to *Unicode locale nonterminals* defined in the grammars of Section 3 Unicode Language and Locale Identifiers <https://unicode.org/reports/tr35/#Unicode_Language_and_Locale_Identifiers>. Each such identifier can also be referred to as a *language tag*, and is in fact a valid language tag as that term is used in BCP 47 <https://www.rfc-editor.org/rfc/bcp/bcp47.txt>. A locale identifier in canonical form as specified in Unicode Technical Standard #35 Part 1 Core, Section 3.2.1 Canonical Unicode Locale Identifiers <https://unicode.org/reports/tr35/#Canonical_Unicode_Locale_Identifiers> is referred to as a "*Unicode canonicalized locale identifier*".

Locale identifiers consist of case-insensitive Unicode Basic Latin alphanumeric *subtags* separated by **"-"** (U+002D HYPHEN-MINUS) characters, with single-character subtags referred to as "*singleton subtags*". Unicode Technical Standard #35 Part 1 Core, Section 3.6 Unicode BCP 47 U Extension <https://unicode.org/reports/tr35/#u_Extension> subtag sequences are used extensively, and the term "*Unicode locale extension sequence*" describes the longest substring of a language tag that can be matched by the **`unicode_locale_extensions`** Unicode locale nonterminal and is not part of a **"-x-…"** private use subtag sequence. It starts with **"-u-"** and includes all immediately following subtags that are not singleton subtags, along with their preceding **"-"** separators. For example, the Unicode locale extension sequence of **"en-US-u-fw-mon-x-u-ex-foobar"** is **"-u-fw-mon"**.

All structurally valid language tags are appropriate for use with the APIs defined by this specification, but implementations are not required to use Unicode Common Locale Data Repository (CLDR <https://cldr.unicode.org>) data for validating them; the set of locales and thus language tags that an implementation supports with adequate localizations is implementation-defined. Intl constructors map requested language tags to locales supported by their respective implementations.

6.2.1 IsStructurallyValidLanguageTag (*locale*)

The abstract operation IsStructurallyValidLanguageTag takes argument *locale* (a String) and returns a Boolean. It determines whether *locale* is a syntactically well-formed language tag. It does not consider whether *locale* conveys any meaningful semantics, nor does it differentiate between aliased subtags and their preferred replacement subtags or require canonical casing or subtag ordering. It performs the following steps when called:

1. Let *lowerLocale* be the ASCII-lowercase of *locale*.
2. If *lowerLocale* cannot be matched by the **`unicode_locale_id`** Unicode locale nonterminal, return **false**.
3. If *lowerLocale* uses any of the backwards compatibility syntax described in Unicode Technical Standard #35 Part 1 Core, Section 3.3 BCP 47 Conformance <https://unicode.org/reports/tr35/#BCP_47_Conformance>, return **false**.
4. Let *languageId* be the longest prefix of *lowerLocale* matched by the **`unicode_language_id`** Unicode locale nonterminal.
5. Let *variants* be GetLocaleVariants(*languageId*).
6. If *variants* is not **undefined**, then
    a. If *variants* contains any duplicate subtags, return **false**.
7. Let *allExtensions* be the suffix of *lowerLocale* following *languageId*.
8. If *allExtensions* contains a substring matched by the **`pu_extensions`** Unicode locale nonterminal, let *extensions* be the prefix of *allExtensions* preceding the longest such substring. Otherwise, let *extensions* be *allExtensions*.
9. If *extensions* is not the empty String, then
    a. If *extensions* contains any duplicate singleton subtags, return **false**.
    b. Let *transformExtension* be the longest substring of *extensions* matched by the **`transformed_extensions`** Unicode locale nonterminal. If there is no such substring, return **true**.
    c. Assert: The substring of *transformExtension* from `0` to 3 is **"-t-"**.
    d. Let *tPrefix* be the substring of *transformExtension* from 3.
    e. Let *tlang* be the longest prefix of *tPrefix* matched by the **`tlang`** Unicode locale nonterminal. If there is no such prefix, return **true**.
    f. Let *tlangRefinements* be the longest suffix of *tlang* following a non-empty prefix matched by the **`unicode_language_subtag`** Unicode locale nonterminal.

   g. If *tlangRefinements* contains any duplicate substrings matched greedily by the **`unicode_variant_subtag`** Unicode locale nonterminal, return **false**.
10. Return **true**.

6.2.2 CanonicalizeUnicodeLocaleId (*locale*)

The abstract operation CanonicalizeUnicodeLocaleId takes argument *locale* (a language tag) and returns a Unicode canonicalized locale identifier. It returns the canonical and case-regularized form of the *locale*. It performs the following steps when called:

1. Let *localeId* be the String value resulting from performing the algorithm to transform *locale* to canonical form per Unicode Technical Standard #35 Part 1 Core, Annex C LocaleId Canonicalization <https://unicode.org/reports/tr35/#LocaleId_Canonicalization> (note that the algorithm begins with canonicalizing syntax only).
2. If *localeId* contains a substring that is a Unicode locale extension sequence, then
   a. Let *extension* be the String value consisting of the substring of the Unicode locale extension sequence within *localeId*.
   b. Let *newExtension* be **"-u"**.
   c. Let *components* be UnicodeExtensionComponents(*extension*).
   d. For each element *attr* of *components*.[[Attributes]], do
      i. Set *newExtension* to the string-concatenation of *newExtension*, **"-"**, and *attr*.
   e. For each Record { [[Key]], [[Value]] } *keyword* of *components*.[[Keywords]], do
      i. Set *newExtension* to the string-concatenation of *newExtension*, **"-"**, and *keyword*.[[Key]].
      ii. If *keyword*.[[Value]] is not the empty String, then
         1. Set *newExtension* to the string-concatenation of *newExtension*, **"-"**, and *keyword*.[[Value]].
   f. Assert: *newExtension* is not equal to **"-u"**.
   g. Set *localeId* to a copy of *localeId* in which the first appearance of substring *extension* has been replaced with *newExtension*.
3. Return *localeId*.

NOTE Step 2 ensures that a Unicode locale extension sequence in the returned language tag contains:

- only the first instance of any attribute duplicated in the input, and
- only the first keyword for a given key in the input.

6.2.3 DefaultLocale ( )

The implementation-defined abstract operation DefaultLocale takes no arguments and returns a Unicode canonicalized locale identifier. The returned String value represents the structurally valid (6.2.1) and canonicalized (6.2.2) language tag for the host environment's current locale. It must not contain a Unicode locale extension sequence.

6.3 Currency Codes

This specification identifies currencies using 3-letter currency codes as defined by ISO 4217. Their canonical form is uppercase.

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 abstract operation IsWellFormedCurrencyCode takes argument *currency* (a String) and returns a Boolean. It verifies that the *currency* argument represents a well-formed 3-letter ISO currency code. It performs the following steps when called:

1. If the length of *currency* is not 3, return **false**.
2. Let *normalized* be the ASCII-uppercase of *currency*.
3. If *normalized* contains any code unit outside of `0x0041` through `0x005A` (corresponding to Unicode characters LATIN CAPITAL LETTER A through LATIN CAPITAL LETTER Z), return **false**.
4. Return **true**.

6.4 AvailableCanonicalCurrencies ( )

The implementation-defined abstract operation AvailableCanonicalCurrencies takes no arguments and returns a List of Strings. The returned List is ordered as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*, and contains unique, well-formed, and upper case canonicalized 3-letter ISO 4217 currency codes, identifying the currencies for which the implementation provides the functionality of Intl.DisplayNames and Intl.NumberFormat objects.

6.5 Time Zone Names

This 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 except as specifically overridden by CanonicalizeTimeZoneName.

A conforming implementation must recognize **"UTC"** and all other Zone and Link names (and **only** 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.

6.5.1 IsValidTimeZoneName (*timeZone*)

The abstract operation IsValidTimeZoneName takes argument *timeZone* (a String) and returns a Boolean. It verifies that the *timeZone* argument represents a valid Zone or Link name of the IANA Time Zone Database. It performs the following steps when called:

1. If one of the Zone or Link names of the IANA Time Zone Database is an ASCII-case-insensitive match for *timeZone*, return **true**.
2. If *timeZone* is an ASCII-case-insensitive match for **"UTC"**, return **true**.
3. Return **false**.

NOTE Any value returned from SystemTimeZoneIdentifier that is not recognized as valid by IsTimeZoneOffsetString must be recognized as valid by IsValidTimeZoneName.

6.5.2 CanonicalizeTimeZoneName (*timeZone*)

The abstract operation CanonicalizeTimeZoneName takes argument *timeZone* (a String value that is a valid time zone name as verified by IsValidTimeZoneName) and returns a String. It returns the canonical and case-regularized form of *timeZone*. It performs the following steps when called:

1. Let *ianaTimeZone* be the String value of the Zone or Link name of the IANA Time Zone Database that is an ASCII-case-insensitive match for *timeZone*.
2. If *ianaTimeZone* is a Link name, set *ianaTimeZone* to the String value of the corresponding Zone name as specified in the file **`backward`** of the IANA Time Zone Database.
3. If *ianaTimeZone* is one of **"Etc/UTC"**, **"Etc/GMT"**, or **"GMT"**, return **"UTC"**.
4. Return *ianaTimeZone*.

6.5.3 AvailableCanonicalTimeZones ( )

The implementation-defined abstract operation AvailableCanonicalTimeZones takes no arguments and returns a List of Strings. The returned List is a sorted List of supported Zone and Link names in the IANA Time Zone Database. It performs the following steps when called:

1. Let *names* be a List of all Zone and Link names in the IANA Time Zone Database that are supported by the implementation.
2. Let *result* be a new empty List.
3. For each element *name* of *names*, do
   a. Assert: IsValidTimeZoneName(*name*) is **true**.
   b. Let *canonical* be CanonicalizeTimeZoneName(*name*).
   c. If *result* does not contain *canonical*, then
      i. Append *canonical* to *result*.
4. Sort *result* in order as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*.
5. Return *result*.

6.6 Measurement Unit Identifiers

This specification identifies measurement units using a *core unit identifier* (or equivalently *core unit ID*) as defined by Unicode Technical Standard #35 Part 2 General, Section 6.2 Unit Identifiers <https://unicode.org/reports/tr35/tr35-general.html#Unit_Identifiers>. Their canonical form is a string containing only Unicode Basic Latin lowercase letters (U+`0061` LATIN SMALL LETTER A through U+`007A` LATIN SMALL LETTER Z) with zero or more medial hyphens (U+`002D` HYPHEN-MINUS).

Only a limited set of core unit identifiers are sanctioned. Attempting to use an unsanctioned core unit identifier results in a **RangeError**.

6.6.1 IsWellFormedUnitIdentifier (*unitIdentifier*)

The abstract operation IsWellFormedUnitIdentifier takes argument *unitIdentifier* (a String) and returns a Boolean. It verifies that the *unitIdentifier* argument represents a well-formed core unit identifier that is either a sanctioned single unit or a complex unit formed by division of two sanctioned single units. It performs the following steps when called:

1. If IsSanctionedSingleUnitIdentifier(*unitIdentifier*) is **true**, then
   a. Return **true**.
2. Let *i* be StringIndexOf(*unitIdentifier*, **"-per-"**, `0`).
3. If *i* is -1 or StringIndexOf(*unitIdentifier*, **"-per-"**, *i* + 1) is not -1, then
   a. Return **false**.
4. Assert: The five-character substring **"-per-"** occurs exactly once in *unitIdentifier*, at index *i*.
5. Let *numerator* be the substring of *unitIdentifier* from `0` to *i*.
6. Let *denominator* be the substring of *unitIdentifier* from *i* + 5.
7. If IsSanctionedSingleUnitIdentifier(*numerator*) and IsSanctionedSingleUnitIdentifier(*denominator*) are both **true**, then
   a. Return **true**.
8. Return **false**.

6.6.2 IsSanctionedSingleUnitIdentifier (*unitIdentifier*)

The abstract operation IsSanctionedSingleUnitIdentifier takes argument *unitIdentifier* (a String) and returns a Boolean. It verifies that the *unitIdentifier* argument is among the single unit identifiers sanctioned in the current version of this specification, which are a subset of the Common Locale Data Repository release 38 unit validity data <https://github.com/unicode-org/cldr/blob/maint/maint-38/common/validity/unit.xml>; the list may grow over time. As discussed in Unicode Technical Standard #35 Part 2 General, Section 6.2 Unit Identifiers <https://uni-

code.org/reports/tr35/tr35-general.html#Unit_Identifiers>, a single unit identifier is a core unit identifier that is not composed of multiplication or division of other unit identifiers. It performs the following steps when called:

1. If *unitIdentifier* is listed in Table 2 below, return **true**.
2. Else, return **false**.

Table 2: Single units sanctioned for use in ECMAScript

Single Unit Identifier
acre
bit
byte
celsius
centimeter
day
degree
fahrenheit
fluid-ounce
foot
gallon
gigabit
gigabyte
gram
hectare
hour
inch
kilobit
kilobyte
kilogram
kilometer
liter
megabit
megabyte
meter
microsecond
mile
mile-scandinavian
milliliter

Table 2: Single units sanctioned for use in ECMAScript
(continued)

Single Unit Identifier
millimeter
millisecond
minute
month
nanosecond
ounce
percent
petabyte
pound
second
stone
terabit
terabyte
week
yard
year

6.6.3 AvailableCanonicalUnits ( )

The abstract operation AvailableCanonicalUnits takes no arguments and returns a List of Strings. The returned List is ordered as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*, and consists of the unique values of simple unit identifiers listed in every row of Table 2, except the header row.

6.7 Numbering System Identifiers

This specification identifies numbering systems using a *numbering system identifier* corresponding with the name referenced by Unicode Technical Standard #35 Part 3 Numbers, Section 1 Numbering Systems <https://unicode.org/reports/tr35/tr35-numbers.html#Numbering_Systems>. Their canonical form is a string containing only Unicode Basic Latin lowercase letters (U+0061 LATIN SMALL LETTER A through U+007A LATIN SMALL LETTER Z).

6.7.1 AvailableCanonicalNumberingSystems ( )

The implementation-defined abstract operation AvailableCanonicalNumberingSystems takes no arguments and returns a List of Strings. The returned List is ordered as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*, and contains unique canonical numbering systems identifiers identifying the numbering systems for which the implementation provides the functionality of Intl.DateTimeFormat, Intl.NumberFormat, and Intl.RelativeTimeFormat objects. The list must include the Numbering System value of every row of Table 14, except the header row.

6.8 Collation Types

This specification identifies collations using a *collation type* as defined by Unicode Technical Standard #35 Part 5 Collation, Section 3.1 Collation Types <https://unicode.org/reports/tr35/tr35-collation.html#Collation_Types>. Their canonical form is a string containing only Unicode Basic Latin lowercase letters (U+0061 LATIN SMALL LETTER A through U+007A LATIN SMALL LETTER Z) with zero or more medial hyphens (U+002D HYPHEN-MINUS).

6.8.1 AvailableCanonicalCollations ( )

The implementation-defined abstract operation AvailableCanonicalCollations takes no arguments and returns a List of Strings. The returned List is ordered as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*, and contains unique canonical collation types identifying the collations for which the implementation provides the functionality of Intl.Collator objects.

6.9 Calendar Types

This specification identifies calendars using a *calendar type* as defined by Unicode Technical Standard #35 Part 4 Dates, Section 2 Calendar Elements <https://unicode.org/reports/tr35/tr35-dates.html#Calendar_Elements>. Their canonical form is a string containing only Unicode Basic Latin lowercase letters (U+0061 LATIN SMALL LETTER A through U+007A LATIN SMALL LETTER Z) with zero or more medial hyphens (U+002D HYPHEN-MINUS).

6.9.1 AvailableCanonicalCalendars ( )

The implementation-defined abstract operation AvailableCanonicalCalendars takes no arguments and returns a List of Strings. The returned List is ordered as if an Array of the same values had been sorted using %Array.prototype.sort% using **undefined** as *comparefn*, and contains unique canonical calendar types identifying the calendars for which the implementation provides the functionality of Intl.DateTimeFormat objects. The list must include **"iso8601"**.

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 2024 Language Specification (ECMA-262 15th Edition, or successor), clause 18.

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 %Object.prototype%.

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 Value Properties of the Intl Object

8.1.1 Intl[@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl"**.

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

8.2 Constructor Properties of the Intl Object

With the exception of Intl.Locale, each of the following constructors is a *service constructor* that creates objects providing locale-sensitive services.

8.2.1 Intl.Collator (. . .)

See 10.

8.2.2 Intl.DateTimeFormat (. . .)

See 11.

8.2.3 Intl.DisplayNames (. . .)

See 12.

8.2.4 Intl.ListFormat (. . .)

See 13.

8.2.5 Intl.Locale (. . .)

See 14.

8.2.6 Intl.NumberFormat (. . .)

See 15.

8.2.7 Intl.PluralRules (. . .)

See 16.

8.2.8 Intl.RelativeTimeFormat (. . .)

See 17.

8.2.9 Intl.Segmenter (. . .)

See 18.

8.3 Function Properties of the Intl Object

8.3.1 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*).

8.3.2 Intl.supportedValuesOf (*key*)

When the **`supportedValuesOf`** method is called with argument *key* , the following steps are taken:

1. Let *key* be ? ToString(*key*).
2. If *key* is **"calendar"**, then
   a. Let *list* be AvailableCanonicalCalendars( ).
3. Else if *key* is **"collation"**, then
   a. Let *list* be AvailableCanonicalCollations( ).
4. Else if *key* is **"currency"**, then
   a. Let *list* be AvailableCanonicalCurrencies( ).
5. Else if *key* is **"numberingSystem"**, then
   a. Let *list* be AvailableCanonicalNumberingSystems( ).
6. Else if *key* is **"timeZone"**, then
   a. Let *list* be AvailableCanonicalTimeZones( ).
7. Else if *key* is **"unit"**, then
   a. Let *list* be AvailableCanonicalUnits( ).
8. Else,
   a. Throw a **RangeError** exception.
9. Return CreateArrayFromList(*list*).

9 Locale and Parameter Negotiation

Service constructors use common patterns to negotiate the requests represented by *locales* and *options* arguments against the actual capabilities of an implementation. That common behaviour is explained here in terms of internal slots describing the capabilities, abstract operations using these internal slots, and specialized data types defined below.

An *Available Locales List* is an arbitrarily-ordered duplicate-free List of language tags, each of which is structurally valid, canonicalized, and lacks a Unicode locale extension sequence. It represents all locales for which the implementation provides functionality within a particular context.

A *language priority list* is a List of structurally valid and canonicalized language tags representing a sequence of locale preferences by descending priority. It corresponds with the term of the same name defined in BCP 47 <https://www.rfc-editor.org/rfc/bcp/bcp47.txt> at RFC 4647 section 2.3 <https://www.rfc-editor.org/rfc/rfc4647.html#section-2.3> but prohibits **"*"** elements and contains only canonicalized contents.

9.1 Internal slots of Service Constructors

Each service constructor has the following internal slots:

- [[AvailableLocales]] is an Available Locales List. It must include the value returned by DefaultLocale. Additionally, for each element with more than one subtag, it must also include a less narrow language tag with the same language subtag and a strict subset of the same following subtags (i.e., omitting one or more) to serve as a potential fallback from ResolveLocale. In particular, each element with a language subtag and a script subtag and a region subtag must be accompanied by another element consisting of only the same language subtag and region subtag but missing the script subtag. For example,
  - If [[AvailableLocales]] contains **"de-DE"**, then it must also contain **"de"** (which might be selected to

satisfy requested locales such as **"de-AT"** and **"de-CH"**).
    - If [[AvailableLocales]] contains **"az-Latn-AZ"**, then it must also contain **"az-AZ"** (which might be selected to satisfy requested locales such as **"az-Cyrl-AZ"** if **"az-Cyrl"** is unavailable).
- [[RelevantExtensionKeys]] is a List of Unicode locale extension sequence keys defined in Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> that are relevant for the functionality of the constructed objects.
- [[SortLocaleData]] and [[SearchLocaleData]] (for Intl.Collator) and [[LocaleData]] (for every other service constructor) are Records. In addition to fields specific to its service constructor, each such Record has a field for each locale contained in [[AvailableLocales]]. The value of each such locale-named field is a Record in which each element of [[RelevantExtensionKeys]] identifies the name of a field whose value is a non-empty List of Strings representing the type values that are supported by the implementation in the relevant locale for the corresponding Unicode locale extension sequence key, with the first element providing the default value for that key in the locale.

NOTE For example, an implementation of DateTimeFormat might include the language tag **"fa-IR"** in its [[AvailableLocales]] internal slot, and must (according to 11.2.3) include the keys **"ca"**, **"hc"**, and **"nu"** in its [[RelevantExtensionKeys]] internal slot. The default calendar for that locale is usually **"persian"**, but an implementation might also support **"gregory"**, **"islamic"**, and **"islamic-civil"**. The Record in the DateTimeFormat [[LocaleData]] internal slot would therefore include a [[fa-IR]] field whose value is a Record like { [[ca]]: « **"persian"**, **"gregory"**, **"islamic"**, **"islamic-civil"** », [[hc]]: « … », [[nu]]: « … » }, along with other locale-named fields having the same value shape but different elements in their Lists.

9.2 Abstract Operations

9.2.1 CanonicalizeLocaleList (*locales*)

The abstract operation CanonicalizeLocaleList takes argument *locales* (an ECMAScript language value) and returns either a normal completion containing a language priority list or a throw completion. It performs the following steps when called:

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 or Type(*locales*) is Object and *locales* has an [[InitializedLocale]] internal slot, then
    a. Let *O* be CreateArrayFromList(« *locales* »).
4. Else,
    a. Let *O* be ? ToObject(*locales*).
5. Let *len* be ? LengthOfArrayLike(*O*).
6. Let *k* be `0`.
7. Repeat, while *k* < *len*,
    a. Let *Pk* be ! ToString(𝔽(*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. If Type(*kValue*) is Object and *kValue* has an [[InitializedLocale]] internal slot, then
            1. Let *tag* be *kValue*.[[Locale]].
        iv. Else,
            1. Let *tag* be ? ToString(*kValue*).
        v. If IsStructurallyValidLanguageTag(*tag*) is **false**, throw a **RangeError** exception.
        vi. Let *canonicalizedTag* be CanonicalizeUnicodeLocaleId(*tag*).
        vii. If *seen* does not contain *canonicalizedTag*, append *canonicalizedTag* to *seen*.
    d. Set *k* to *k* + 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 LookupMatchingLocaleByPrefix (*availableLocales*, *requestedLocales*)

The abstract operation LookupMatchingLocaleByPrefix takes arguments *availableLocales* (an Available Locales List) and *requestedLocales* (a language priority list) and returns a Record with fields [[locale]] (a Unicode canonicalized locale identifier) and [[extension]] (a Unicode locale extension sequence or EMPTY) or **undefined**. It determines the best element of *availableLocales* for satisfying *requestedLocales* using the lookup algorithm defined in BCP 47 <https://www.rfc-editor.org/rfc/bcp/bcp47.txt> at RFC 4647 section 3.4 <https://www.rfc-editor.org/rfc/rfc4647.html#section-3.4>, ignoring Unicode locale extension sequences. If a non-default match is found, it returns a Record with a [[locale]] field containing the matching language tag from *availableLocales* and an [[extension]] field containing the Unicode locale extension sequence of the corresponding element of *requestedLocales* (or EMPTY if requested language tag has no such sequence). It performs the following steps when called:

1. For each element *locale* of *requestedLocales*, do
   a. Let *extension* be EMPTY.
   b. If *locale* contains a Unicode locale extension sequence, then
      i. Set *extension* to the Unicode locale extension sequence of *locale*.
      ii. Set *locale* to the String value that is *locale* with any Unicode locale extension sequences removed.
   c. Let *prefix* be *locale*.
   d. Repeat, while *prefix* is not the empty String,
      i. If *availableLocales* contains *prefix*, return the Record { [[locale]]: *prefix*, [[extension]]: *extension* }.
      ii. If *prefix* contains **"-"** (code unit `0x002D` HYPHEN-MINUS), let *pos* be the index into *prefix* of the last occurrence of **"-"**; else let *pos* be `0`.
      iii. Repeat, while *pos* ≥ 2 and the substring of *prefix* from *pos* - 2 to *pos* - 1 is **"-"**,
         1. Set *pos* to *pos* - 2.
      iv. Set *prefix* to the substring of *prefix* from `0` to *pos*.
2. Return **undefined**.

NOTE When a requested locale includes a Unicode Technical Standard #35 Part 1 Core BCP 47 T Extension <https://unicode.org/reports/tr35/#BCP47_T_Extension> subtag sequence, the truncation in this algorithm may temporarily generate invalid language tags. However, none of them will be returned because *availableLocales* contains only valid language tags.

9.2.3 LookupMatchingLocaleByBestFit (*availableLocales*, *requestedLocales*)

The implementation-defined abstract operation LookupMatchingLocaleByBestFit takes arguments *availableLocales* (an Available Locales List) and *requestedLocales* (a language priority list) and returns a Record with fields [[locale]] (a Unicode canonicalized locale identifier) and [[extension]] (a Unicode locale extension sequence or EMPTY), or **undefined**. It determines the best element of *availableLocales* for satisfying *requestedLocales*, ignoring Unicode locale extension sequences. The algorithm is implementation dependent, but should produce results that a typical user of the requested locales would consider at least as good as those produced by the LookupMatchingLocaleByPrefix algorithm. If a non-default match is found, it returns a Record with a [[locale]] field containing the matching language tag from *availableLocales* and an [[extension]] field containing the Unicode locale extension sequence of the corresponding element of *requestedLocales* (or EMPTY if requested language tag has no such sequence).

9.2.4 UnicodeExtensionComponents (*extension*)

The abstract operation UnicodeExtensionComponents takes argument *extension* (a Unicode locale extension sequence) and returns a Record with fields [[Attributes]] and [[Keywords]]. It deconstructs *extension* into a List of unique attributes and a List of keywords with unique keys. Any repeated appearance of an attribute or keyword key after the first is ignored. It performs the following steps when called:

1. Assert: The ASCII-lowercase of *extension* is *extension*.
2. Assert: The substring of *extension* from `0` to 3 is **"-u-"**.
3. Let *attributes* be a new empty List.
4. Let *keywords* be a new empty List.
5. Let *keyword* be **undefined**.
6. Let *size* be the length of *extension*.
7. Let *k* be 3.
8. Repeat, while *k* < *size*,
    a. Let *e* be StringIndexOf(*extension*, **"-"**, *k*).
    b. If *e* = -1, let *len* be *size* - *k*; else let *len* be *e* - *k*.
    c. Let *subtag* be the substring of *extension* from *k* to *k* + *len*.
    d. NOTE: A keyword is a sequence of subtags in which the first is a key of length 2 and any subsequent ones (if present) have length in the inclusive interval from 3 to 8, collectively constituting a value along with their medial **"-"** separators. An attribute is a single subtag with length in the inclusive interval from 3 to 8 that precedes all keywords.
    e. Assert: *len* ≥ 2.
    f. If *keyword* is **undefined** and *len* ≠ 2, then
        i. If *subtag* is not an element of *attributes*, append *subtag* to *attributes*.
    g. Else if *len* = 2, then
        i. Set *keyword* to the Record { [[Key]]: *subtag*, [[Value]]: **""** }.
        ii. If *keywords* does not contain an element whose [[Key]] is *keyword*.[[Key]], append *keyword* to *keywords*.
    h. Else if *keyword*.[[Value]] is the empty String, then
        i. Set *keyword*.[[Value]] to *subtag*.
    i. Else,
        i. Set *keyword*.[[Value]] to the string-concatenation of *keyword*.[[Value]], **"-"**, and *subtag*.
    j. Set *k* to *k* + *len* + 1.
9. Return the Record { [[Attributes]]: *attributes*, [[Keywords]]: *keywords* }.

9.2.5 InsertUnicodeExtensionAndCanonicalize (*locale*, *extension*)

The abstract operation InsertUnicodeExtensionAndCanonicalize takes arguments *locale* (a Unicode canonicalized locale identifier) and *extension* (a Unicode locale extension sequence) and returns a Unicode canonicalized locale identifier. It incorporates *extension* into *locale* and returns the canonicalized result. It performs the following steps when called:

1. Assert: *locale* does not contain a Unicode locale extension sequence.
2. Let *privateIndex* be StringIndexOf(*locale*, **"-x-"**, `0`).
3. If *privateIndex* = -1, then
    a. Let *newLocale* be the string-concatenation of *locale* and *extension*.
4. Else,
    a. Let *preExtension* be the substring of *locale* from `0` to *privateIndex*.
    b. Let *postExtension* be the substring of *locale* from *privateIndex*.
    c. Let *newLocale* be the string-concatenation of *preExtension*, *extension*, and *postExtension*.
5. Assert: IsStructurallyValidLanguageTag(*newLocale*) is **true**.
6. Return CanonicalizeUnicodeLocaleId(*newLocale*).

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

The abstract operation ResolveLocale takes arguments *availableLocales* (an Available Locales List), *requestedLocales* (a language priority list), *options* (a Record), *relevantExtensionKeys* (a List of Strings), and *localeData* (a Record) and returns a Record. It performs "lookup" as defined in BCP 47 <https://www.rfc-editor.org/rfc/bcp/bcp47.txt> at RFC 4647 section 3 <https://www.rfc-editor.org/rfc/rfc4647.html#section-3>, determining the best element of *availableLocales* for satisfying *requestedLocales* using either the LookupMatchingLocaleByBestFit algorithm or LookupMatchingLocaleByPrefix algorithm as specified by *options*.[[localeMatcher]], ignoring Unicode locale extension sequences, and returns a representation of the match that also includes corresponding data from *localeData* and a resolved value for each element of *relevantExtensionKeys* (defaulting to data from the matched locale, superseded by data from the requested Unicode locale extension sequence if present and then by data from *options* if present). If the matched element from *requestedLocales* contains a Unicode locale extension sequence, it is copied onto the language tag in the [[Locale]] field of the returned Record, omitting any **`keyword`** Unicode locale nonterminal whose **key** value is not contained within *relevantExtensionKeys* or **`type`** value is superseded by a different value from *options*. It performs the following steps when called:

1. Let *matcher* be *options*.[[localeMatcher]].
2. If *matcher* is **"lookup"**, then
    a. Let *r* be LookupMatchingLocaleByPrefix(*availableLocales*, *requestedLocales*).
3. Else,
    a. Let *r* be LookupMatchingLocaleByBestFit(*availableLocales*, *requestedLocales*).
4. If *r* is **undefined**, set *r* to the Record { [[locale]]: DefaultLocale(), [[extension]]: EMPTY }.
5. Let *foundLocale* be *r*.[[locale]].
6. Let *foundLocaleData* be *localeData*.[[<*foundLocale*>]].
7. Assert: Type(*foundLocaleData*) is Record.
8. Let *result* be a new Record.
9. Set *result*.[[LocaleData]] to *foundLocaleData*.
10. If *r*.[[extension]] is not EMPTY, then
    a. Let *components* be UnicodeExtensionComponents(*r*.[[extension]]).
    b. Let *keywords* be *components*.[[Keywords]].
11. Let *supportedExtension* be **"-u"**.
12. For each element *key* of *relevantExtensionKeys*, do
    a. Let *keyLocaleData* be *foundLocaleData*.[[<*key*>]].
    b. Assert: Type(*keyLocaleData*) is List.
    c. Let *value* be *keyLocaleData*[`0`].
    d. Assert: Type(*value*) is either String or Null.
    e. Let *supportedExtensionAddition* be **""**.
    f. If *r*.[[extension]] is not EMPTY, then
        i. If *keywords* contains an element whose [[Key]] is the same as *key*, then
            1. Let *entry* be the element of *keywords* whose [[Key]] is the same as *key*.
            2. Let *requestedValue* be *entry*.[[Value]].
            3. If *requestedValue* is not the empty String, then
                a. If *keyLocaleData* contains *requestedValue*, then
                    i. Set *value* to *requestedValue*.
                    ii. Set *supportedExtensionAddition* to the string-concatenation of **"-"**, *key*, **"-"**, and *value*.
            4. Else if *keyLocaleData* contains **"true"**, then
                a. Set *value* to **"true"**.
                b. Set *supportedExtensionAddition* to the string-concatenation of **"-"** and *key*.
    g. 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 Type(*optionsValue*) is String, then
            1. Assert: The ASCII-lowercase of *key* is *key*.
            2. Let *optionsUValue* be the ASCII-lowercase of *optionsValue*.
            3. Set *optionsValue* to the String value resulting from canonicalizing *optionsUValue* as a value of key *key* per Unicode Technical Standard #35 Part 1 Core, Annex C LocaleId Canonicalization Section 5 Canonicalizing Syntax, Processing LocaleIds <https://unicode.org/reports/tr35/#processing-localeids>.
            4. NOTE: It is recommended that implementations use the 'u' extension data in **`common/bcp47`** provided by the Common Locale Data Repository (available at https://cldr.unicode.org/).
            5. If *optionsValue* is the empty String, then
                a. Set *optionsValue* to **"true"**.
        iv. If SameValue(*optionsValue*, *value*) is **false** and *keyLocaleData* contains *optionsValue*, then
            1. Set *value* to *optionsValue*.
            2. Set *supportedExtensionAddition* to **""**.
    h. Set *result*.[[<*key*>]] to *value*.

      i. Set *supportedExtension* to the string-concatenation of *supportedExtension* and *supportedExtensionAddition*.
13. If *supportedExtension* is **"-u"**, then
    a. Set *result*.[[Locale]] to *foundLocale*.
14. Else,
    a. Set *result*.[[Locale]] to InsertUnicodeExtensionAndCanonicalize(*foundLocale*, *supportedExtension*).
15. Return *result*.

9.2.7 FilterLocales (*availableLocales*, *requestedLocales*, *options*)

The abstract operation FilterLocales takes arguments *availableLocales* (an Available Locales List), *requestedLocales* (a language priority list), and *options* (an ECMAScript language value) and returns either a normal completion containing a List of Unicode canonicalized locale identifiers or a throw completion. It performs "filtering" as defined in BCP 47 <https://www.rfc-editor.org/rfc/bcp/bcp47.txt> at RFC 4647 section 3 <https://www.rfc-editor.org/rfc/rfc4647.html#section-3>, returning the elements of *requestedLocales* for which *availableLocales* contains a matching locale when using either the LookupMatchingLocaleByBestFit algorithm or LookupMatchingLocaleByPrefix algorithm as specified in *options*, preserving their relative order. It performs the following steps when called:

1. Set *options* to ? CoerceOptionsToObject(*options*).
2. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, STRING, « **"lookup"**, **"best fit"** », **"best fit"**).
3. Let *subset* be a new empty List.
4. For each element *locale* of *requestedLocales*, do
   a. Let *noExtensionsLocale* be the String value that is *locale* with any Unicode locale extension sequences removed.
   b. If *matcher* is **"lookup"**, then
      i. Let *match* be LookupMatchingLocaleByPrefix(*availableLocales*, *noExtensionsLocale*).
   c. Else,
      i. Let *match* be LookupMatchingLocaleByBestFit(*availableLocales*, *noExtensionsLocale*).
   d. If *match* is not **undefined**, append *locale* to *subset*.
5. Return CreateArrayFromList(*subset*).

9.2.8 GetOptionsObject (*options*)

The abstract operation GetOptionsObject takes argument *options* (an ECMAScript language value) and returns either a normal completion containing an Object or a throw completion. It returns an Object suitable for use with GetOption, either *options* itself or a default empty Object. It throws a **TypeError** if *options* is not **undefined** and not an Object. It performs the following steps when called:

1. If *options* is **undefined**, then
   a. Return OrdinaryObjectCreate(**null**).
2. If Type(*options*) is Object, then
   a. Return *options*.
3. Throw a **TypeError** exception.

9.2.9 CoerceOptionsToObject (*options*)

The abstract operation CoerceOptionsToObject takes argument *options* (an ECMAScript language value) and returns either a normal completion containing an Object or a throw completion. It coerces *options* into an Object suitable for use with GetOption, defaulting to an empty Object. Because it coerces non-null primitive values into objects, its use is discouraged for new functionality in favour of GetOptionsObject. It performs the following steps when called:

1. If *options* is **undefined**, then
   a. Return OrdinaryObjectCreate(**null**).
2. Return ? ToObject(*options*).

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

The abstract operation GetOption takes arguments *options* (an Object), *property* (a property key), *type* (BOOLEAN or STRING), *values* (EMPTY or a List of ECMAScript language values), and *default* (REQUIRED or an ECMAScript language value) and returns either a normal completion containing an ECMAScript language value or a throw completion. It extracts the value of the specified property of *options*, converts it to the required *type*, checks whether it is allowed by *values* if *values* is not EMPTY, and substitutes *default* if the value is **undefined**. It performs the following steps when called:

1. Let *value* be ? Get(*options*, *property*).
2. If *value* is **undefined**, then
   a. If *default* is REQUIRED, throw a **RangeError** exception.
   b. Return *default*.
3. If *type* is BOOLEAN, then
   a. Set *value* to ToBoolean(*value*).
4. Else,
   a. Assert: *type* is STRING.
   b. Set *value* to ? ToString(*value*).
5. If *values* is not EMPTY and *values* does not contain *value*, throw a **RangeError** exception.
6. Return *value*.

9.2.11 GetBooleanOrStringNumberFormatOption (*options*, *property*, *stringValues*, *fallback*)

The abstract operation GetBooleanOrStringNumberFormatOption takes arguments *options* (an Object), *property* (a property key), *stringValues* (a List of Strings), and *fallback* (an ECMAScript language value) and returns either a normal completion containing either a Boolean, String, or *fallback*, or a throw completion. It extracts the value of the property named *property* from the provided *options* object. It returns *fallback* if that value is **undefined**, **true** if that value is **true**, **false** if that value coerces to **false**, and otherwise coerces it to a String and returns the result if it is allowed by *stringValues*. It performs the following steps when called:

1. Let *value* be ? Get(*options*, *property*).
2. If *value* is **undefined**, return *fallback*.
3. If *value* is **true**, return **true**.
4. If ToBoolean(*value*) is **false**, return **false**.
5. Set *value* to ? ToString(*value*).
6. If *stringValues* does not contain *value*, throw a **RangeError** exception.
7. Return *value*.

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

The abstract operation DefaultNumberOption takes arguments *value* (an ECMAScript language value), *minimum* (an integer), *maximum* (an integer), and *fallback* (an integer or **undefined**) and returns either a normal completion containing either an integer or **undefined**, or a throw completion. It converts *value* to an integer, checks whether it is in the allowed range, and fills in a *fallback* value if necessary. It performs the following steps when called:

1. If *value* is **undefined**, return *fallback*.
2. Set *value* to ? ToNumber(*value*).
3. If *value* is not finite or $\mathbb{R}$(*value*) < *minimum* or $\mathbb{R}$(*value*) > *maximum*, throw a **RangeError** exception.
4. Return floor($\mathbb{R}$(*value*)).

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

The abstract operation GetNumberOption takes arguments *options* (an Object), *property* (a String), *minimum* (an integer), *maximum* (an integer), and *fallback* (an integer or **undefined**) and returns either a normal completion containing either an integer or **undefined**, or a throw completion. It extracts the value of the property named *property* from the provided *options* object, converts it to an integer, checks whether it is in the allowed range, and fills in a *fallback* value if necessary. It performs the following steps when called:

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

9.2.14 PartitionPattern (*pattern*)

The abstract operation PartitionPattern takes argument *pattern* (a String) and returns a List of Records with fields [[Type]] (a String) and [[Value]] (a String or **undefined**). The [[Value]] field will be a String value if [[Type]] is **"literal"**, and **undefined** otherwise. The syntax of the abstract pattern strings is an implementation detail and is not exposed to users of ECMA-402. It performs the following steps when called:

1. Let *result* be a new empty List.
2. Let *beginIndex* be StringIndexOf(*pattern*, **"{"**, `0`).
3. Let *endIndex* be `0`.
4. Let *nextIndex* be `0`.
5. Let *length* be the number of code units in *pattern*.
6. Repeat, while *beginIndex* is an integer index into *pattern*,
   a. Set *endIndex* to StringIndexOf(*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 the Record { [[Type]]: **"literal"**, [[Value]]: *literal* } to *result*.
   d. Let *p* be the substring of *pattern* from position *beginIndex*, exclusive, to position *endIndex*, exclusive.
   e. Append the Record { [[Type]]: *p*, [[Value]]: **undefined** } to *result*.
   f. Set *nextIndex* to *endIndex* + 1.
   g. Set *beginIndex* to StringIndexOf(*pattern*, **"{"**, *nextIndex*).
7. 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 the Record { [[Type]]: **"literal"**, [[Value]]: *literal* } to *result*.
8. Return *result*.

10 Collator Objects

10.1 The Intl.Collator Constructor

The Intl.Collator constructor is the *%Intl.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 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 %Intl.Collator%.[[RelevantExtensionKeys]] contains **"kn"**, then
   a. Append [[Numeric]] to *internalSlotsList*.
4. If %Intl.Collator%.[[RelevantExtensionKeys]] contains **"kf"**, then
   a. Append [[CaseFirst]] to *internalSlotsList*.
5. Let *collator* be ? OrdinaryCreateFromConstructor(*newTarget*, **"%Intl.Collator.prototype%"**, *internalSlotsList*).
6. Return ? InitializeCollator(*collator*, *locales*, *options*).

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

The abstract operation InitializeCollator takes arguments *collator* (an Intl.Collator), *locales* (an ECMAScript language value), and *options* (an ECMAScript language value) and returns either a normal completion containing *collator* or a throw completion. It initializes *collator* as a Collator object. It performs the following steps when called:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. Set *options* to ? CoerceOptionsToObject(*options*).
3. Let *usage* be ? GetOption(*options*, **"usage"**, STRING, « **"sort"**, **"search"** », **"sort"**).
4. Set *collator*.[[Usage]] to *usage*.
5. If *usage* is **"sort"**, then
    a. Let *localeData* be %Intl.Collator%.[[SortLocaleData]].
6. Else,
    a. Let *localeData* be %Intl.Collator%.[[SearchLocaleData]].
7. Let *opt* be a new Record.
8. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, STRING, « **"lookup"**, **"best fit"** », **"best fit"**).
9. Set *opt*.[[localeMatcher]] to *matcher*.
10. Let *collation* be ? GetOption(*options*, **"collation"**, STRING, EMPTY, **undefined**).
11. If *collation* is not **undefined**, then
    a. If *collation* cannot be matched by the `type` Unicode locale nonterminal, throw a **RangeError** exception.
12. Set *opt*.[[co]] to *collation*.
13. Let *numeric* be ? GetOption(*options*, **"numeric"**, BOOLEAN, EMPTY, **undefined**).
14. If *numeric* is not **undefined**, then
    a. Set *numeric* to ! ToString(*numeric*).
15. Set *opt*.[[kn]] to *numeric*.
16. Let *caseFirst* be ? GetOption(*options*, **"caseFirst"**, STRING, « **"upper"**, **"lower"**, **"false"** », **undefined**).
17. Set *opt*.[[kf]] to *caseFirst*.
18. Let *relevantExtensionKeys* be %Intl.Collator%.[[RelevantExtensionKeys]].
19. Let *r* be ResolveLocale(%Intl.Collator%.[[AvailableLocales]], *requestedLocales*, *opt*, *relevantExtensionKeys*, *localeData*).
20. Set *collator*.[[Locale]] to *r*.[[Locale]].
21. Set *collation* to *r*.[[co]].
22. If *collation* is **null**, set *collation* to **"default"**.
23. Set *collator*.[[Collation]] to *collation*.
24. If *relevantExtensionKeys* contains **"kn"**, then
    a. Set *collator*.[[Numeric]] to SameValue(*r*.[[kn]], **"true"**).
25. If *relevantExtensionKeys* contains **"kf"**, then
    a. Set *collator*.[[CaseFirst]] to *r*.[[kf]].
26. Let *resolvedLocaleData* be *r*.[[LocaleData]].
27. Let *sensitivity* be ? GetOption(*options*, **"sensitivity"**, STRING, « **"base"**, **"accent"**, **"case"**, **"variant"** », **undefined**).
28. If *sensitivity* is **undefined**, then
    a. If *usage* is **"sort"**, then
        i. Set *sensitivity* to **"variant"**.
    b. Else,
        i. Set *sensitivity* to *resolvedLocaleData*.[[sensitivity]].
29. Set *collator*.[[Sensitivity]] to *sensitivity*.
30. Let *defaultIgnorePunctuation* be *resolvedLocaleData*.[[ignorePunctuation]].
31. Let *ignorePunctuation* be ? GetOption(*options*, **"ignorePunctuation"**, BOOLEAN, EMPTY, *defaultIgnorePunctuation*).
32. Set *collator*.[[IgnorePunctuation]] to *ignorePunctuation*.
33. Return *collator*.

NOTE The collation associated with the **"search"** value for the **"usage"** option should only be used to find matching strings, since this collation is not guaranteed to be in any particular order. This behaviour is described in Unicode Technical Standard #35 Part 1 Core, Unicode Collation Identifier <https://unicode.org/reports/tr35/#UnicodeCollationIdentifier> and Unicode Technical Standard #10 Unicode Collation Algorithm, Searching and Matching <https://unicode.org/reports/tr10/#Searching>.

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 %Intl.Collator.prototype%.

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 with arguments *locales* and *options*, the following steps are taken:

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

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 **"kf"** and **"kn"**, and must not include any other elements.

NOTE Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> 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, **"kf"** for case first, **"kh"** for hiragana quaternary, **"kk"** for normalization, **"kn"** for numeric, **"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, for all locale values *locale*:

- The first element of [[SortLocaleData]].[[<*locale*>]].[[co]] and [[SearchLocaleData]].[[<*locale*>]].[[co]] must be **null**.
- 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"**.
- [[SearchLocaleData]].[[<*locale*>]] and [[SortLocaleData]].[[<*locale*>]] must have an [[ignorePunctuation]] field with a Boolean value.

10.3 Properties of the Intl.Collator Prototype Object

The Intl.Collator prototype object is itself an ordinary object. *%Intl.Collator.prototype%* 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 %Intl.Collator%.

10.3.2 Intl.Collator.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.Collator"**.

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 the **this** value.
2. Perform ? RequireInternalSlot(*collator*, [[InitializedCollator]]).
3. 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*.
4. 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**$_{\mathbb{F}}$.

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

The implementation-defined abstract operation CompareStrings takes arguments *collator* (an Intl.Collator), *x* (a String), and *y* (a String) and returns a Number, but not **NaN**. The returned Number represents the result of an implementation-defined locale-sensitive String comparison of *x* with *y*. The result is intended to correspond with a sort order of String values according to the effective locale and collation options of *collator*, and will be negative when *x* is ordered before *y*, positive when *x* is ordered after *y*, and zero in all other cases (representing no relative ordering between *x* and *y*). String values must be interpreted as UTF-16 code unit sequences as described in es2024, 6.1.4, 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.

Behaviour as described below depends upon locale-sensitive identification of the sequence of collation elements for a string, in particular "base letters", and different base letters always compare as unequal (causing the strings containing them to also compare as unequal). Results of comparing variations of the same base letter with different case, diacritic marks, or potentially other aspects further depends upon *collator*.[[Sensitivity]] as follows:

Table 3: Effects of Collator Sensitivity

[[Sensitivity]]	Description	"a" vs. "á"	"a" vs. "A"
"base"	Characters with the same base letter do not compare as unequal, regardless of differences in case and/or diacritic marks.	equal	equal
"accent"	Characters with the same base letter compare as unequal only if they differ in accents and/or other diacritic marks, regardless of differences in case.	not equal	equal
"case"	Characters with the same base letter compare as unequal only if they differ in case, regardless of differences in accents and/or other diacritic marks.	equal	not equal
"variant"	Characters with the same base letter compare as unequal if they differ in case, diacritic marks, and/or potentially other differences.	not equal	not equal

NOTE 1 The mapping from input code points to base letters can include arbitrary contractions, expansions, and collisions, including those that apply special treatment to certain characters with diacritic marks. For example, in Swedish, "ö" is a base letter that differs from "o", and "v" and "w" are considered to be the same base letter. In Slovak, "ch" is a single base letter, and in English, "æ" is a sequence of base letters starting with "a" and ending with "e".

If *collator*.[[IgnorePunctuation]] is **true**, then punctuation is ignored (e.g., strings that differ only in punctuation compare as equal).

For the interpretation of options settable through locale extension keys, see Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_>.

The actual return values are implementation-defined to permit encoding additional information in them, but this operation for any given *collator*, when considered as a function of *x* and *y*, is required to be a consistent comparator defining a total ordering on the set of all Strings. This operation is also required to recognize and honour canonical equivalence according to the Unicode Standard, including returning **+0**$_{\mathbb{F}}$ when comparing distinguishable Strings that are canonically equivalent.

NOTE 2 It is recommended that the CompareStrings abstract operation be implemented following Unicode Technical Standard #10: Unicode Collation Algorithm <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 https://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 options computed during initialization of the object.

1. Let *collator* be the **this** value.
2. Perform ? RequireInternalSlot(*collator*, [[InitializedCollator]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. 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 *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 %Intl.Collator%.[[RelevantExtensionKeys]] does not contain *extensionKey*, then
         1. Set *v* to **undefined**.
   d. If *v* is not **undefined**, then
      i. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
5. Return *options*.

Table 4: 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 %Intl.Collator.prototype%.

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 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> 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 4 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 DateTimeFormat Objects

11.1 The Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor is the *%Intl.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.

11.1.1 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 ? CreateDateTimeFormat(*newTarget*, *locales*, *options*, ANY, DATE).
3. If the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Let *this* be the **this** value.
   b. Return ? ChainDateTimeFormat(*dateTimeFormat*, NewTarget, *this*).
4. Return *dateTimeFormat*.

NORMATIVE OPTIONAL

11.1.1.1 ChainDateTimeFormat (*dateTimeFormat*, *newTarget*, *this*)

The abstract operation ChainDateTimeFormat takes arguments *dateTimeFormat* (an Intl.DateTimeFormat), *newTarget* (an ECMAScript language value), and *this* (an ECMAScript language value) and returns either a normal completion containing an Object or a throw completion. It performs the following steps when called:

1. If *newTarget* is **undefined** and ? OrdinaryHasInstance(%Intl.DateTimeFormat%, *this*) is **true**, then
   a. Perform ? DefinePropertyOrThrow(*this*, %Intl%.[[FallbackSymbol]], PropertyDescriptor{ [[Value]]: *dateTimeFormat*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
   b. Return *this*.
2. Return *dateTimeFormat*.

11.1.2 CreateDateTimeFormat (*newTarget*, *locales*, *options*, *required*, *defaults*)

The abstract operation CreateDateTimeFormat takes arguments *newTarget* (a constructor), *locales* (an ECMAScript language value), *options* (an ECMAScript language value), *required* (DATE, TIME, or ANY), and *defaults* (DATE, TIME, or ALL) and returns either a normal completion containing a DateTimeFormat object or a throw completion. It performs the following steps when called:

1. Let *dateTimeFormat* be ? OrdinaryCreateFromConstructor(*newTarget*, **"%Intl.DateTimeFormat.prototype%"**, « [[InitializedDateTimeFormat]], [[Locale]], [[Calendar]], [[NumberingSystem]], [[TimeZone]], [[Weekday]], [[Era]], [[Year]], [[Month]], [[Day]], [[DayPeriod]], [[Hour]], [[Minute]], [[Second]], [[FractionalSecondDigits]], [[TimeZoneName]], [[HourCycle]], [[DateStyle]], [[TimeStyle]], [[Pattern]], [[RangePatterns]], [[BoundFormat]] »).
2. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
3. Set *options* to ? CoerceOptionsToObject(*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 *calendar* be ? GetOption(*options*, **"calendar"**, STRING, EMPTY, **undefined**).
8. If *calendar* is not **undefined**, then
   a. If *calendar* cannot be matched by the `type` Unicode locale nonterminal, throw a **RangeError** exception.

9. Set *opt*.[[ca]] to *calendar*.
10. Let *numberingSystem* be ? GetOption(*options*, **"numberingSystem"**, STRING, EMPTY, **undefined**).
11. If *numberingSystem* is not **undefined**, then
    a. If *numberingSystem* cannot be matched by the **type** Unicode locale nonterminal, throw a **RangeError** exception.
12. Set *opt*.[[nu]] to *numberingSystem*.
13. Let *hour12* be ? GetOption(*options*, **"hour12"**, BOOLEAN, EMPTY, **undefined**).
14. Let *hourCycle* be ? GetOption(*options*, **"hourCycle"**, STRING, « **"h11"**, **"h12"**, **"h23"**, **"h24"** », **undefined**).
15. If *hour12* is not **undefined**, then
    a. Set *hourCycle* to **null**.
16. Set *opt*.[[hc]] to *hourCycle*.
17. Let *r* be ResolveLocale(%Intl.DateTimeFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.DateTimeFormat%.[[RelevantExtensionKeys]], %Intl.DateTimeFormat%.[[LocaleData]]).
18. Set *dateTimeFormat*.[[Locale]] to *r*.[[Locale]].
19. Let *resolvedCalendar* be *r*.[[ca]].
20. Set *dateTimeFormat*.[[Calendar]] to *resolvedCalendar*.
21. Set *dateTimeFormat*.[[NumberingSystem]] to *r*.[[nu]].
22. Let *resolvedLocaleData* be *r*.[[LocaleData]].
23. If *hour12* is **true**, then
    a. Let *hc* be *resolvedLocaleData*.[[hourCycle12]].
24. Else if *hour12* is **false**, then
    a. Let *hc* be *resolvedLocaleData*.[[hourCycle24]].
25. Else,
    a. Assert: *hour12* is **undefined**.
    b. Let *hc* be *r*.[[hc]].
    c. If *hc* is **null**, set *hc* to *resolvedLocaleData*.[[hourCycle]].
26. Set *dateTimeFormat*.[[HourCycle]] to *hc*.
27. Let *timeZone* be ? Get(*options*, **"timeZone"**).
28. If *timeZone* is **undefined**, then
    a. Set *timeZone* to SystemTimeZoneIdentifier().
29. Else,
    a. Set *timeZone* to ? ToString(*timeZone*).
30. If IsTimeZoneOffsetString(*timeZone*) is **true**, then
    a. Let *parseResult* be ParseText(StringToCodePoints(*timeZone*), *UTCOffset*).
    b. Assert: *parseResult* is a Parse Node.
    c. If *parseResult* contains more than one *MinuteSecond* Parse Node, throw a **RangeError** exception.
    d. Let *offsetNanoseconds* be ParseTimeZoneOffsetString(*timeZone*).
    e. Let *offsetMinutes* be *offsetNanoseconds* / (6×10^{10}).
    f. Assert: *offsetMinutes* is an integer.
    g. Set *timeZone* to FormatOffsetTimeZoneIdentifier(*offsetMinutes*).
31. Else if IsValidTimeZoneName(*timeZone*) is **true**, then
    a. Set *timeZone* to CanonicalizeTimeZoneName(*timeZone*).
32. Else,
    a. Throw a **RangeError** exception.
33. Set *dateTimeFormat*.[[TimeZone]] to *timeZone*.
34. Let *formatOptions* be a new Record.
35. Set *formatOptions*.[[hourCycle]] to *hc*.
36. Let *hasExplicitFormatComponents* be **false**.
37. For each row of Table 7, except the header row, in table order, do
    a. Let *prop* be the name given in the Property column of the current row.
    b. If *prop* is **"fractionalSecondDigits"**, then
        i. Let *value* be ? GetNumberOption(*options*, **"fractionalSecondDigits"**, 1, 3, **undefined**).
    c. Else,
        i. Let *values* be a List whose elements are the strings given in the Values column of the current row.
        ii. Let *value* be ? GetOption(*options*, *prop*, STRING, *values*, **undefined**).
    d. Set *formatOptions*.[[<*prop*>]] to *value*.
    e. If *value* is not **undefined**, then
        i. Set *hasExplicitFormatComponents* to **true**.
38. Let *formatMatcher* be ? GetOption(*options*, **"formatMatcher"**, STRING, « **"basic"**, **"best fit"** », **"best fit"**).

39. Let *dateStyle* be ? GetOption(*options*, **"dateStyle"**, STRING, « **"full"**, **"long"**, **"medium"**, **"short"** », **undefined**).
40. Set *dateTimeFormat*.[[DateStyle]] to *dateStyle*.
41. Let *timeStyle* be ? GetOption(*options*, **"timeStyle"**, STRING, « **"full"**, **"long"**, **"medium"**, **"short"** », **undefined**).
42. Set *dateTimeFormat*.[[TimeStyle]] to *timeStyle*.
43. If *dateStyle* is **undefined** and *timeStyle* is **undefined**, then
    a. Let *needDefaults* be **true**.
    b. If *required* is DATE or ANY, then
        i. For each property name *prop* of « **"weekday"**, **"year"**, **"month"**, **"day"** », do
            1. Let *value* be *formatOptions*.[[<*prop*>]].
            2. If *value* is not **undefined**, set *needDefaults* to **false**.
    c. If *required* is TIME or ANY, then
        i. For each property name *prop* of « **"dayPeriod"**, **"hour"**, **"minute"**, **"second"**, **"fractionalSecondDigits"** », do
            1. Let *value* be *formatOptions*.[[<*prop*>]].
            2. If *value* is not **undefined**, set *needDefaults* to **false**.
    d. If *needDefaults* is **true** and *defaults* is either DATE or ALL, then
        i. For each property name *prop* of « **"year"**, **"month"**, **"day"** », do
            1. Set *formatOptions*.[[<*prop*>]] to **"numeric"**.
    e. If *needDefaults* is **true** and *defaults* is either TIME or ALL, then
        i. For each property name *prop* of « **"hour"**, **"minute"**, **"second"** », do
            1. Set *formatOptions*.[[<*prop*>]] to **"numeric"**.
    f. Let *formats* be *resolvedLocaleData*.[[formats]].[[<*resolvedCalendar*>]].
    g. If *formatMatcher* is **"basic"**, then
        i. Let *bestFormat* be BasicFormatMatcher(*formatOptions*, *formats*).
    h. Else,
        i. Let *bestFormat* be BestFitFormatMatcher(*formatOptions*, *formats*).
44. Else,
    a. If *hasExplicitFormatComponents* is **true**, then
        i. Throw a **TypeError** exception.
    b. If *required* is DATE and *timeStyle* is not **undefined**, then
        i. Throw a **TypeError** exception.
    c. If *required* is TIME and *dateStyle* is not **undefined**, then
        i. Throw a **TypeError** exception.
    d. Let *styles* be *resolvedLocaleData*.[[styles]].[[<*resolvedCalendar*>]].
    e. Let *bestFormat* be DateTimeStyleFormat(*dateStyle*, *timeStyle*, *styles*).
45. For each row of Table 7, except the header row, in table order, do
    a. Let *prop* be the name given in the Property column of the current row.
    b. If *bestFormat* has a field [[<*prop*>]], then
        i. Let *p* be *bestFormat*.[[<*prop*>]].
        ii. Set *dateTimeFormat*'s internal slot whose name is the Internal Slot column of the current row to *p*.
46. If *dateTimeFormat*.[[Hour]] is **undefined**, then
    a. Set *dateTimeFormat*.[[HourCycle]] to **undefined**.
47. If *dateTimeFormat*.[[HourCycle]] is **"h11"** or **"h12"**, then
    a. Let *pattern* be *bestFormat*.[[pattern12]].
    b. Let *rangePatterns* be *bestFormat*.[[rangePatterns12]].
48. Else,
    a. Let *pattern* be *bestFormat*.[[pattern]].
    b. Let *rangePatterns* be *bestFormat*.[[rangePatterns]].
49. Set *dateTimeFormat*.[[Pattern]] to *pattern*.
50. Set *dateTimeFormat*.[[RangePatterns]] to *rangePatterns*.
51. Return *dateTimeFormat*.

11.1.3 FormatOffsetTimeZoneIdentifier (*offsetMinutes*)

The abstract operation FormatOffsetTimeZoneIdentifier takes argument *offsetMinutes* (an integer) and returns a String. It formats a UTC offset, in minutes, into a UTC offset string formatted like ±HH:MM. It performs the following steps when called:

1. If *offsetMinutes* ≥ `0`, let *sign* be the code unit `0x002B` (PLUS SIGN); otherwise, let *sign* be the code unit `0x002D` (HYPHEN-MINUS).
2. Let *absoluteMinutes* be abs(*offsetMinutes*).
3. Let *hours* be floor(*absoluteMinutes* / `60`).
4. Let *minutes* be *absoluteMinutes* modulo `60`.
5. Return the string-concatenation of *sign*, ToZeroPaddedDecimalString(*hours*, 2), the code unit `0x003A` (COLON), and ToZeroPaddedDecimalString(*minutes*, 2).

11.2 Properties of the Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor has the following properties:

11.2.1 Intl.DateTimeFormat.prototype

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

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

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

11.2.3 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 « **"ca"**, **"hc"**, **"nu"** ».

> NOTE 1 Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_Type_Definitions> describes four locale extension keys that are relevant to date and time formatting: **"ca"** for calendar, **"hc"** for hour cycle, **"nu"** for numbering system (of formatted numbers), and **"tz"** for time zone. 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, for all locale values *locale*:

- [[LocaleData]].[[<*locale*>]].[[nu]] must be a List that does not include the values **"native"**, **"traditio"**, or **"finance"**.
- [[LocaleData]].[[<*locale*>]].[[hc]] must be « **null**, **"h11"**, **"h12"**, **"h23"**, **"h24"** ».
- [[LocaleData]].[[<*locale*>]].[[hourCycle]] must be a String value equal to **"h11"**, **"h12"**, **"h23"**, or **"h24"**.
- [[LocaleData]].[[<*locale*>]].[[hourCycle12]] must be a String value equal to **"h11"** or **"h12"**.
- [[LocaleData]].[[<*locale*>]].[[hourCycle24]] must be a String value equal to **"h23"** or **"h24"**.
- [[LocaleData]].[[<*locale*>]] must have a [[formats]] field. This [[formats]] field must be a Record with [[<*calendar*>]] fields for all calendar values *calendar*. The value of this field must be a list of records, each of which has a subset of the fields shown in Table 7, where each field must have one of the values specified for

the field in Table 7. 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, fractionalSecondDigits
  - weekday, year, month, day, hour, minute, second
  - weekday, year, month, day
  - year, month, day
  - year, month
  - month, day
  - hour, minute, second, fractionalSecondDigits
  - hour, minute, second
  - hour, minute
  - dayPeriod, hour
  - dayPeriod, hour, minute, second
  - dayPeriod, hour, minute

Each of the records must also have the following fields:

1. 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 **"}"**.
2. 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 at least one of the substrings **"{ampm}"** or **"{dayPeriod}"**.
3. If the record has a [[year]] field, the [[pattern]] and [[pattern12]] values may contain the substrings **"{yearName}"** and **"{relatedYear}"**.
4. A [[rangePatterns]] field with a Record value:
   - The [[rangePatterns]] record may have any of the fields in Table 5, where each field represents a range pattern and its value is a Record.
     - The name of the field indicates the largest calendar element that must be different between the start and end dates in order to use this range pattern. For example, if the field name is [[Month]], it contains the range pattern that should be used to format a date range where the era and year values are the same, but the month value is different.
     - The record will contain the following fields:
       - A subset of the fields shown in the Property column of Table 7, where each field must have one of the values specified for that field in the Values column of Table 7. All fields required to format a date for any of the [[PatternParts]] records must be present.
       - A [[PatternParts]] field whose value is a list of Records each representing a part of the range pattern. Each record contains a [[Pattern]] field and a [[Source]] field. The [[Pattern]] field's value is a String of the same format as the regular date pattern String. The [[Source]] field is one of the String values **"shared"**, **"startRange"**, or **"endRange"**. It indicates which of the range's dates should be formatted using the value of the [[Pattern]] field.
   - The [[rangePatterns]] record must have a [[Default]] field which contains the default range pattern used when the specific range pattern is not available. Its value is a list of records with the same structure as the other fields in the [[rangePatterns]] record.
5. If the record has an [[hour]] field, it must also have a [[rangePatterns12]] field. Its value is similar to the Record in [[rangePatterns]], but it uses a String similar to [[pattern12]] for each part of the range pattern.
6. If the record has a [[year]] field, the [[rangePatterns]] and [[rangePatterns12]] fields may contain range patterns where the [[Pattern]] values may contain the substrings **"{yearName}"** and **"{relatedYear}"**.

- [[LocaleData]].[[<*locale*>]] must have a [[styles]] field. The [[styles]] field must be a Record with [[<*calendar*>]] fields for all calendar values *calendar*. The calendar records must contain [[DateFormat]], [[TimeFormat]], [[DateTimeFormat]] and [[DateTimeRangeFormat]] fields, the value of these fields are Records, where each of which has [[full]], [[long]], [[medium]] and [[short]] fields. For [[DateFormat]] and [[TimeFormat]], the value of these fields must be a record, which has a subset of the fields shown in Table 7, where each field must have one of the values specified for the field in Table 7. Each of the records must also have the following fields:
  1. 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 **"}"**.
  2. 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 at least one of the substrings **"{ampm}"** or **"{dayPeriod}"**.
  3. A [[rangePatterns]] field that contains a record similar to the one described in the [[formats]] field.

4. If the record has an [[hour]] field, it must also have a [[rangePatterns12]] field. Its value is similar to the record in [[rangePatterns]] but it uses a string similar to [[pattern12]] for each range pattern.

For [[DateTimeFormat]], the field value must be a string pattern which contains the strings **"{0}"** and **"{1}"**. For [[DateTimeRangeFormat]] the value of these fields must be a nested record which also has [[full]], [[long]], [[medium]] and [[short]] fields. The [[full]], [[long]], [[medium]] and [[short]] fields in the enclosing record refer to the date style of the range pattern, while the fields in the nested record refers to the time style of the range pattern. The value of these fields in the nested record is a record with a [[rangePatterns]] field and a [[rangePatterns12]] field which are similar to the [[rangePatterns]] and [rangePatterns12]] fields in [[DateFormat]] and [[TimeFormat]].

NOTE 2 For example, an implementation might include the following record as part of its English locale data:

- [[hour]]: **"numeric"**
- [[minute]]: **"numeric"**
- [[pattern]]: **"{hour}:{minute}"**
- [[pattern12]]: **"{hour}:{minute} {ampm}"**
- [[rangePatterns]]:
  - [[Hour]]:
    - [[hour]]: **"numeric"**
    - [[minute]]: **"numeric"**
    - [[PatternParts]]:
      - {[[Source]]: **"startRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
      - {[[Source]]: **"endRange"**, [[Pattern]]: **"{hour}:{minute}"**}
  - [[Minute]]:
    - [[hour]]: **"numeric"**
    - [[minute]]: **"numeric"**
    - [[PatternParts]]:
      - {[[Source]]: **"startRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
      - {[[Source]]: **"endRange"**, [[Pattern]]: **"{hour}:{minute}"**}
  - [[Default]]:
    - [[year]]: **"2-digit"**
    - [[month]]: **"numeric"**
    - [[day]]: **"numeric"**
    - [[hour]]: **"numeric"**
    - [[minute]]: **"numeric"**
    - [[PatternParts]]:
      - {[[Source]]: **"startRange"**, [[Pattern]]: **"{day}/{month}/{year}, {hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
      - {[[Source]]: **"endRange"**, [[Pattern]]: **"{day}/{month}/{year}, {hour}:{minute}"**}
- [[rangePatterns12]]:
  - [[Hour]]:
    - [[hour]]: **"numeric"**
    - [[minute]]: **"numeric"**
    - [[PatternParts]]:
      - {[[Source]]: **"startRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
      - {[[Source]]: **"endRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" {ampm}"**}
  - [[Minute]]:
    - [[hour]]: **"numeric"**
    - [[minute]]: **"numeric"**
    - [[PatternParts]]:
      - {[[Source]]: **"startRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
      - {[[Source]]: **"endRange"**, [[Pattern]]: **"{hour}:{minute}"**}
      - {[[Source]]: **"shared"**, [[Pattern]]: **" {ampm}"**}
  - [[Default]]:

  - [[year]]: **"2-digit"**
  - [[month]]: **"numeric"**
  - [[day]]: **"numeric"**
  - [[hour]]: **"numeric"**
  - [[minute]]: **"numeric"**
  - [[PatternParts]]:
    - {[[Source]]: **"startRange"**, [[Pattern]]: **"{day}/{month}/{year}, {hour}:{minute}{ampm}"**}
    - {[[Source]]: **"shared"**, [[Pattern]]: **" – "**}
    - {[[Source]]: **"endRange"**, [[Pattern]]: **"{day}/{month}/{year}, {hour}:{minute}{ampm}"**}

NOTE 3 It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at https://cldr.unicode.org/).

Table 5: Range pattern fields

Range Pattern Field	Pattern String Field
[[Era]]	**"era"**
[[Year]]	**"year"**
[[Month]]	**"month"**
[[Day]]	**"day"**
[[AmPm]]	**"ampm"**
[[DayPeriod]]	**"dayPeriod"**
[[Hour]]	**"hour"**
[[Minute]]	**"minute"**
[[Second]]	**"second"**
[[FractionalSecondDigits]]	**"fractionalSecondDigits"**

11.3 Properties of the Intl.DateTimeFormat Prototype Object

The Intl.DateTimeFormat prototype object is itself an ordinary object. *%Intl.DateTimeFormat.prototype%* 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.

11.3.1 Intl.DateTimeFormat.prototype.constructor

The initial value of `Intl.DateTimeFormat.prototype.constructor` is %Intl.DateTimeFormat%.

11.3.2 Intl.DateTimeFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.DateTimeFormat"**.

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

11.3.3 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 the **this** value.
2. If the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Set *dtf* to ? UnwrapDateTimeFormat(*dtf*).
3. Perform ? RequireInternalSlot(*dtf*, [[InitializedDateTimeFormat]]).
4. If *dtf*.[[BoundFormat]] is **undefined**, then
   a. Let *F* be a new built-in function object as defined in DateTime Format Functions (11.5.4).
   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.

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

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

1. Let *dtf* be the **this** value.
2. Perform ? RequireInternalSlot(*dtf*, [[InitializedDateTimeFormat]]).
3. If *date* is **undefined**, then
   a. Let *x* be ! Call(%Date.now%, **undefined**).
4. Else,
   a. Let *x* be ? ToNumber(*date*).
5. Return ? FormatDateTimeToParts(*dtf*, *x*).

11.3.5 Intl.DateTimeFormat.prototype.formatRange (*startDate*, *endDate*)

When the **`formatRange`** method is called with arguments *startDate* and *endDate*, the following steps are taken:

1. Let *dtf* be **this** value.
2. Perform ? RequireInternalSlot(*dtf*, [[InitializedDateTimeFormat]]).
3. If *startDate* is **undefined** or *endDate* is **undefined**, throw a **TypeError** exception.
4. Let *x* be ? ToNumber(*startDate*).
5. Let *y* be ? ToNumber(*endDate*).
6. Return ? FormatDateTimeRange(*dtf*, *x*, *y*).

11.3.6 Intl.DateTimeFormat.prototype.formatRangeToParts (*startDate*, *endDate*)

When the **`formatRangeToParts`** method is called with arguments *startDate* and *endDate*, the following steps are taken:

1. Let *dtf* be **this** value.
2. Perform ? RequireInternalSlot(*dtf*, [[InitializedDateTimeFormat]]).
3. If *startDate* is **undefined** or *endDate* is **undefined**, throw a **TypeError** exception.
4. Let *x* be ? ToNumber(*startDate*).
5. Let *y* be ? ToNumber(*endDate*).
6. Return ? FormatDateTimeRangeToParts(*dtf*, *x*, *y*).

11.3.7 Intl.DateTimeFormat.prototype.resolvedOptions ( )

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

1. Let *dtf* be the **this** value.
2. If the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Set *dtf* to ? UnwrapDateTimeFormat(*dtf*).
3. Perform ? RequireInternalSlot(*dtf*, [[InitializedDateTimeFormat]]).
4. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
5. For each row of Table 6, except the header row, in table order, do
   a. Let *v* be the value of *dtf*'s internal slot whose name is the Internal Slot value of the current row.
   b. Let *p* be the Property value of the current row.
   c. If the Internal Slot value of the current row is an Internal Slot value in Table 7, then
      i. If *dtf*.[[DateStyle]] is not **undefined** or *dtf*.[[TimeStyle]] is not **undefined**, then
         1. Set *v* to **undefined**.
   d. If *v* is not **undefined**, then
      i. If there is a Conversion value in the current row, then
         1. Let *conversion* be the Conversion value of the current row.
         2. If *conversion* is HOUR12, then
            a. If *v* is **"h11"** or **"h12"**, set *v* to **true**. Otherwise, set *v* to **false**.
         3. Else,
            a. Assert: *conversion* is NUMBER.
            b. Set *v* to $\mathbb{F}$(*v*).
      ii. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 6: Resolved Options of DateTimeFormat Instances

Internal Slot	Property	Conversion
[[Locale]]	**"locale"**	
[[Calendar]]	**"calendar"**	
[[NumberingSystem]]	**"numberingSystem"**	
[[TimeZone]]	**"timeZone"**	
[[HourCycle]]	**"hourCycle"**	
[[HourCycle]]	**"hour12"**	HOUR12
[[Weekday]]	**"weekday"**	
[[Era]]	**"era"**	
[[Year]]	**"year"**	
[[Month]]	**"month"**	
[[Day]]	**"day"**	
[[DayPeriod]]	**"dayPeriod"**	
[[Hour]]	**"hour"**	
[[Minute]]	**"minute"**	
[[Second]]	**"second"**	
[[FractionalSecondDigits]]	**"fractionalSecondDigits"**	NUMBER
[[TimeZoneName]]	**"timeZoneName"**	

Table 6: Resolved Options of DateTimeFormat Instances
(continued)

Internal Slot	Property	Conversion
[[DateStyle]]	**"dateStyle"**	
[[TimeStyle]]	**"timeStyle"**	

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 API, the **"timeZone"** property will be the identifier of the host environment's 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.

11.4 Properties of Intl.DateTimeFormat Instances

Intl.DateTimeFormat instances are ordinary objects that inherit properties from %Intl.DateTimeFormat.prototype%.

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 representing the Unicode Calendar Identifier <https://unicode.org/reports/tr35/#UnicodeCalendarIdentifier> used for formatting.
- [[NumberingSystem]] is a String value representing the Unicode Number System Identifier <https://unicode.org/reports/tr35/#UnicodeNumberSystemIdentifier> used for formatting.
- [[TimeZone]] is a String value used for formatting that is either a time zone identifier from the IANA Time Zone Database or a UTC offset in ISO 8601 extended format.
- [[Weekday]], [[Era]], [[Year]], [[Month]], [[Day]], [[DayPeriod]], [[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 7, indicating how the component should be presented in the formatted output.
- [[FractionalSecondDigits]] is either **undefined** or a positive, non-zero integer indicating the fraction digits to be used for fractional seconds. Numbers will be rounded or padded with trailing zeroes if necessary.
- [[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**.
- [[DateStyle]], [[TimeStyle]] are each either **undefined**, or a String value with values **"full"**, **"long"**, **"medium"**, or **"short"**.
- [[Pattern]] is a String value as described in 11.2.3.
- [[RangePatterns]] is a Record as described in 11.2.3.

Finally, Intl.DateTimeFormat instances have a [[BoundFormat]] internal slot that caches the function returned by the format accessor (11.3.3).

11.5 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 7: 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"**
[[DayPeriod]]	**"dayPeriod"**	**"narrow"**, **"short"**, **"long"**
[[Hour]]	**"hour"**	**"2-digit"**, **"numeric"**
[[Minute]]	**"minute"**	**"2-digit"**, **"numeric"**
[[Second]]	**"second"**	**"2-digit"**, **"numeric"**
[[FractionalSecondDigits]]	**"fractionalSecondDigits"**	1, 2, 3
[[TimeZoneName]]	**"timeZoneName"**	**"short"**, **"long"**, **"shortOffset"**, **"longOffset"**, **"shortGeneric"**, **"longGeneric"**

11.5.1 DateTimeStyleFormat (*dateStyle*, *timeStyle*, *styles*)

The abstract operation DateTimeStyleFormat takes arguments *dateStyle* (**"full"**, **"long"**, **"medium"**, **"short"**, or **undefined**), *timeStyle* (**"full"**, **"long"**, **"medium"**, **"short"**, or **undefined**), and *styles* (a Record) and returns a Record. *styles* is a record from %Intl.DateTimeFormat%.[[LocaleData]].[[<*locale*>]].[[styles]].[[<*calendar*>]] for some locale *locale* and calendar *calendar*. It returns the appropriate format record for date time formatting based on the parameters. It performs the following steps when called:

1. Assert: *dateStyle* is not **undefined** or *timeStyle* is not **undefined**.
2. If *timeStyle* is not **undefined**, then
   a. Assert: *timeStyle* is one of **"full"**, **"long"**, **"medium"**, or **"short"**.
   b. Let *timeFormat* be *styles*.[[TimeFormat]].[[<*timeStyle*>]].
3. If *dateStyle* is not **undefined**, then
   a. Assert: *dateStyle* is one of **"full"**, **"long"**, **"medium"**, or **"short"**.
   b. Let *dateFormat* be *styles*.[[DateFormat]].[[<*dateStyle*>]].
4. If *dateStyle* is not **undefined** and *timeStyle* is not **undefined**, then
   a. Let *format* be a new Record.
   b. Add to *format* all fields from *dateFormat* except [[pattern]] and [[rangePatterns]].
   c. Add to *format* all fields from *timeFormat* except [[pattern]], [[rangePatterns]], [[pattern12]], and [[rangePatterns12]], if present.
   d. Let *connector* be *styles*.[[DateTimeFormat]].[[<*dateStyle*>]].
   e. Let *pattern* be the string *connector* with the substring **"{0}"** replaced with *timeFormat*.[[pattern]] and the substring **"{1}"** replaced with *dateFormat*.[[pattern]].
   f. Set *format*.[[pattern]] to *pattern*.
   g. If *timeFormat* has a [[pattern12]] field, then
      i. Let *pattern12* be the string *connector* with the substring **"{0}"** replaced with *timeFormat*.[[pattern12]] and the substring **"{1}"** replaced with *dateFormat*.[[pattern]].
      ii. Set *format*.[[pattern12]] to *pattern12*.
   h. Let *dateTimeRangeFormat* be *styles*.[[DateTimeRangeFormat]].[[<*dateStyle*>]].[[<*timeStyle*>]].

    i. Set *format*.[[rangePatterns]] to *dateTimeRangeFormat*.[[rangePatterns]].
    j. If *dateTimeRangeFormat* has a [[rangePatterns12]] field, then
        i. Set *format*.[[rangePatterns12]] to *dateTimeRangeFormat*.[[rangePatterns12]].
    k. Return *format*.
5. If *timeStyle* is not **undefined**, then
    a. Return *timeFormat*.
6. Assert: *dateStyle* is not **undefined**.
7. Return *dateFormat*.

11.5.2 BasicFormatMatcher (*options*, *formats*)

The abstract operation BasicFormatMatcher takes arguments *options* (a Record) and *formats* (a List of Records) and returns a Record. It performs the following steps when called:

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 *offsetPenalty* be 1.
8. Let *bestScore* be -∞.
9. Let *bestFormat* be **undefined**.
10. Assert: Type(*formats*) is List.
11. For each element *format* of *formats*, do
    a. Let *score* be 0.
    b. For each row of Table 7, except the header row, in table order, do
        i. Let *property* be the name given in the Property column of the current row.
        ii. If *options* has a field [[<*property*>]], let *optionsProp* be *options*.[[<*property*>]]; else let *optionsProp* be **undefined**.
        iii. If *format* has a field [[<*property*>]], let *formatProp* be *format*.[[<*property*>]]; else let *formatProp* be **undefined**.
        iv. If *optionsProp* is **undefined** and *formatProp* is not **undefined**, then
            1. Set *score* to *score* - *additionPenalty*.
        v. Else if *optionsProp* is not **undefined** and *formatProp* is **undefined**, then
            1. Set *score* to *score* - *removalPenalty*.
        vi. Else if *property* is **"timeZoneName"**, then
            1. If *optionsProp* is **"short"** or **"shortGeneric"**, then
                a. If *formatProp* is **"shortOffset"**, set *score* to *score* - *offsetPenalty*.
                b. Else if *formatProp* is **"longOffset"**, set *score* to *score* - (*offsetPenalty* + *shortMorePenalty*).
                c. Else if *optionsProp* is **"short"** and *formatProp* is **"long"**, set *score* to *score* - *shortMorePenalty*.
                d. Else if *optionsProp* is **"shortGeneric"** and *formatProp* is **"longGeneric"**, set *score* to _score - *shortMorePenalty*.
                e. Else if *optionsProp* ≠ *formatProp*, set *score* to *score* - *removalPenalty*.
            2. Else if *optionsProp* is **"shortOffset"** and *formatProp* is **"longOffset"**, then
                a. Set *score* to *score* - *shortMorePenalty*.
            3. Else if *optionsProp* is **"long"** or **"longGeneric"**, then
                a. If *formatProp* is **"longOffset"**, set *score* to *score* - *offsetPenalty*.
                b. Else if *formatProp* is **"shortOffset"**, set *score* to *score* - (*offsetPenalty* + *longLessPenalty*).
                c. Else if *optionsProp* is **"long"** and *formatProp* is **"short"**, set *score* to *score* - *longLessPenalty*.
                d. Else if *optionsProp* is **"longGeneric"** and *formatProp* is **"shortGeneric"**, set *score* to *score* - *longLessPenalty*.
                e. Else if *optionsProp* ≠ *formatProp*, set *score* to *score* - *removalPenalty*.
            4. Else if *optionsProp* is **"longOffset"** and *formatProp* is **"shortOffset"**, then
                a. Set *score* to *score* - *longLessPenalty*.
            5. Else if *optionsProp* ≠ *formatProp*, then

a. Set *score* to *score* - *removalPenalty*.

vii. Else if *optionsProp* ≠ *formatProp*, then

1. If *property* is **"fractionalSecondDigits"**, then
   a. Let *values* be « 1, 2, 3 ».
2. Else,
   a. Let *values* be « **"2-digit"**, **"numeric"**, **"narrow"**, **"short"**, **"long"** ».
3. Let *optionsPropIndex* be the index of *optionsProp* within *values*.
4. Let *formatPropIndex* be the index of *formatProp* within *values*.
5. Let *delta* be max(min(*formatPropIndex* - *optionsPropIndex*, 2), -2).
6. If *delta* = 2, set *score* to *score* - *longMorePenalty*.
7. Else if *delta* = 1, set *score* to *score* - *shortMorePenalty*.
8. Else if *delta* = -1, set *score* to *score* - *shortLessPenalty*.
9. Else if *delta* = -2, set *score* to *score* - *longLessPenalty*.

c. If *score* > *bestScore*, then
   i. Set *bestScore* to *score*.
   ii. Set *bestFormat* to *format*.

12. Return *bestFormat*.

11.5.3 BestFitFormatMatcher (*options*, *formats*)

The implementation-defined abstract operation BestFitFormatMatcher takes arguments *options* (a Record) and *formats* (a List of Records) and returns a Record. It returns 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.

11.5.4 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 $\mathbf{1}_{\mathbb{F}}$.

11.5.5 FormatDateTimePattern (*dateTimeFormat*, *patternParts*, *x*, *rangeFormatOptions*)

The abstract operation FormatDateTimePattern takes arguments *dateTimeFormat* (an Intl.DateTimeFormat), *patternParts* (a List of Records as returned by PartitionPattern), *x* (a Number), and *rangeFormatOptions* (a range pattern Record as used in [[rangePattern]], or **undefined**) and returns either a normal completion containing a List of Records with fields [[Type]] (a String) and [[Value]] (a String), or a throw completion. It interprets *x* as a time value as specified in es2024, 21.4.1.1, and creates the corresponding parts according *pattern* and to the effective locale and the formatting options of *dateTimeFormat* and *rangeFormatOptions*. It performs the following steps when called:

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 OrdinaryObjectCreate(**null**).
5. Perform ! CreateDataPropertyOrThrow(*nfOptions*, **"useGrouping"**, **false**).
6. Let *nf* be ! Construct(%Intl.NumberFormat%, « *locale*, *nfOptions* »).
7. Let *nf2Options* be OrdinaryObjectCreate(**null**).
8. Perform ! CreateDataPropertyOrThrow(*nf2Options*, **"minimumIntegerDigits"**, $\mathbf{2}_{\mathbb{F}}$).
9. Perform ! CreateDataPropertyOrThrow(*nf2Options*, **"useGrouping"**, **false**).

10. Let *nf2* be ! Construct(%Intl.NumberFormat%, « *locale*, *nf2Options* »).
11. Let *fractionalSecondDigits* be *dateTimeFormat*.[[FractionalSecondDigits]].
12. If *fractionalSecondDigits* is not **undefined**, then
    a. Let *nf3Options* be OrdinaryObjectCreate(**null**).
    b. Perform ! CreateDataPropertyOrThrow(*nf3Options*, **"minimumIntegerDigits"**, 𝔽(*fractionalSecondDigits*)).
    c. Perform ! CreateDataPropertyOrThrow(*nf3Options*, **"useGrouping"**, **false**).
    d. Let *nf3* be ! Construct(%Intl.NumberFormat%, « *locale*, *nf3Options* »).
13. Let *tm* be ToLocalTime(ℤ(ℝ(*x*) × 10^6), *dateTimeFormat*.[[Calendar]], *dateTimeFormat*.[[TimeZone]]).
14. Let *result* be a new empty List.
15. For each Record { [[Type]], [[Value]] } *patternPart* of *patternParts*, do
    a. Let *p* be *patternPart*.[[Type]].
    b. If *p* is **"literal"**, then
        i. Append the Record { [[Type]]: **"literal"**, [[Value]]: *patternPart*.[[Value]] } to *result*.
    c. Else if *p* is equal to **"fractionalSecondDigits"**, then
        i. Assert: *fractionalSecondDigits* is not **undefined**.
        ii. Let *v* be *tm*.[[Millisecond]].
        iii. Set *v* to floor(*v* × $10^{(fractionalSecondDigits - 3)}$).
        iv. Let *fv* be FormatNumeric(*nf3*, *v*).
        v. Append the Record { [[Type]]: **"fractionalSecond"**, [[Value]]: *fv* } to *result*.
    d. Else if *p* is equal to **"dayPeriod"**, then
        i. Let *f* be *dateTimeFormat*.[[DayPeriod]].
        ii. Let *fv* be a String value representing the day period of *tm* in the form given by *f*; the String value depends upon the implementation and the effective locale of *dateTimeFormat*.
        iii. Append the Record { [[Type]]: *p*, [[Value]]: *fv* } to *result*.
    e. Else if *p* is equal to **"timeZoneName"**, then
        i. Let *f* be *dateTimeFormat*.[[TimeZoneName]].
        ii. Let *v* be *dateTimeFormat*.[[TimeZone]].
        iii. Let *fv* be a String value representing *v* in the form given by *f*; the String value depends upon the implementation and the effective locale of *dateTimeFormat*. The String value may also depend on the value of the [[InDST]] field of *tm* if *f* is **"short"**, **"long"**, **"shortOffset"**, or **"longOffset"**. If the implementation does not have such a localized representation of *v*, then use the String value of *v* itself.
        iv. Append the Record { [[Type]]: *p*, [[Value]]: *fv* } to *result*.
    f. Else if *p* matches one of the values in the Property column of Table 7, then
        i. If *rangeFormatOptions* is not **undefined**, let *f* be the value of *rangeFormatOptions*'s field whose name matches *p*.
        ii. Else, let *f* be the value of *dateTimeFormat*'s internal slot whose name matches the value in the Internal Slot column of the matching row.
        iii. Let *v* be the value of *tm*'s field whose name is the Internal Slot column of the matching row.
        iv. If *p* is **"year"** and *v* ≤ 0, set *v* to 1 - *v*.
        v. If *p* is **"month"**, set *v* to *v* + 1.
        vi. If *p* is **"hour"** and *dateTimeFormat*.[[HourCycle]] is **"h11"** or **"h12"**, then
            1. Set *v* to *v* modulo 12.
            2. If *v* is 0 and *dateTimeFormat*.[[HourCycle]] is **"h12"**, set *v* to 12.
        vii. If *p* is **"hour"** and *dateTimeFormat*.[[HourCycle]] is **"h24"**, then
            1. If *v* is 0, set *v* to 24.
        viii. If *f* is **"numeric"**, then
            1. Let *fv* be FormatNumeric(*nf*, *v*).
        ix. Else if *f* is **"2-digit"**, then
            1. Let *fv* be FormatNumeric(*nf2*, *v*).
            2. If the **"length"** property of *fv* is greater than 2, set *fv* to the substring of *fv* containing the last two characters.
        x. Else if *f* is **"narrow"**, **"short"**, or **"long"**, then
            1. Let *fv* be a String value representing *v* in the form given by *f*; the String value depends upon the implementation and the effective locale and calendar of *dateTimeFormat*. If *p* is **"month"** and *rangeFormatOptions* is **undefined**, then the String value may also depend on whether *dateTimeFormat*.[[Day]] is **undefined**. If *p* is **"month"** and *rangeFormatOptions* is not **undefined**, then the String value may also depend on whether *rangeFormatOptions*.[[day]] is **undefined**. If *p* is **"era"** and *rangeFormatOptions* is **undefined**, then the String value may also

depend on whether *dateTimeFormat*.[[Era]] is **undefined**. If *p* is **"era"** and *rangeFormatOptions* is not **undefined**, then the String value may also depend on whether *rangeFormatOptions*.[[era]] is **undefined**. If the implementation does not have such a localized representation of *v*, then use ! ToString(*v*).

xi. Append the Record { [[Type]]: *p*, [[Value]]: *fv* } to *result*.

g. 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. Append the Record { [[Type]]: **"dayPeriod"**, [[Value]]: *fv* } to *result*.
h. Else if *p* is equal to **"relatedYear"**, then
   i. Let *v* be *tm*.[[RelatedYear]].
   ii. Let *fv* be FormatNumeric(*nf*, *v*).
   iii. Append the Record { [[Type]]: **"relatedYear"**, [[Value]]: *fv* } to *result*.
i. Else if *p* is equal to **"yearName"**, then
   i. Let *v* be *tm*.[[YearName]].
   ii. Let *fv* be an implementation and locale dependent String value representing *v*.
   iii. Append the Record { [[Type]]: **"yearName"**, [[Value]]: *fv* } to *result*.
j. Else,
   i. Let *unknown* be an implementation-, locale-, and numbering system-dependent String based on *x* and *p*.
   ii. Append the Record { [[Type]]: **"unknown"**, [[Value]]: *unknown* } to *result*.

16. Return *result*.

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

11.5.6 PartitionDateTimePattern (*dateTimeFormat*, *x*)

The abstract operation PartitionDateTimePattern takes arguments *dateTimeFormat* (an Intl.DateTimeFormat) and *x* (a Number) and returns either a normal completion containing a List of Records with fields [[Type]] (a String) and [[Value]] (a String), or a throw completion. It interprets *x* as a time value as specified in es2024, 21.4.1.1, and creates the corresponding parts according to the effective locale and the formatting options of *dateTimeFormat*. It performs the following steps when called:

1. Let *patternParts* be PartitionPattern(*dateTimeFormat*.[[Pattern]]).
2. Let *result* be ? FormatDateTimePattern(*dateTimeFormat*, *patternParts*, *x*, **undefined**).
3. Return *result*.

11.5.7 FormatDateTime (*dateTimeFormat*, *x*)

The abstract operation FormatDateTime takes arguments *dateTimeFormat* (an Intl.DateTimeFormat) and *x* (a Number) and returns either a normal completion containing a String or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionDateTimePattern(*dateTimeFormat*, *x*).
2. Let *result* be the empty String.
3. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

11.5.8 FormatDateTimeToParts (*dateTimeFormat*, *x*)

The abstract operation FormatDateTimeToParts takes arguments *dateTimeFormat* (an Intl.DateTimeFormat) and *x* (a Number) and returns either a normal completion containing an Array or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionDateTimePattern(*dateTimeFormat*, *x*).
2. Let *result* be ! ArrayCreate(0).
3. Let *n* be 0.
4. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
   d. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString(𝔽(*n*)), *O*).
   e. Increment *n* by 1.
5. Return *result*.

11.5.9 PartitionDateTimeRangePattern (*dateTimeFormat*, *x*, *y*)

The abstract operation PartitionDateTimeRangePattern takes arguments *dateTimeFormat* (an Intl.DateTime-Format), *x* (a Number), and *y* (a Number) and returns either a normal completion containing a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Source]] (a String), or a throw completion. It interprets *x* and *y* as time values as specified in es2024, 21.4.1.1, and creates the corresponding parts according to the effective locale and the formatting options of *dateTimeFormat*. It performs the following steps when called:

1. Set *x* to TimeClip(*x*).
2. If *x* is **NaN**, throw a **RangeError** exception.
3. Set *y* to TimeClip(*y*).
4. If *y* is **NaN**, throw a **RangeError** exception.
5. Let *tm1* be ToLocalTime(ℤ(ℝ(*x*) × 10^{6}), *dateTimeFormat*.[[Calendar]], *dateTimeFormat*.[[TimeZone]]).
6. Let *tm2* be ToLocalTime(ℤ(ℝ(*y*) × 10^{6}), *dateTimeFormat*.[[Calendar]], *dateTimeFormat*.[[TimeZone]]).
7. Let *rangePatterns* be *dateTimeFormat*.[[RangePatterns]].
8. Let *selectedRangePattern* be **undefined**.
9. Let *relevantFieldsEqual* be **true**.
10. Let *checkMoreFields* be **true**.
11. For each row of Table 5, except the header row, in table order, do
    a. Let *fieldName* be the name given in the Range Pattern Field column of the current row.
    b. If *rangePatterns* has a field [[<*fieldName*>]], let *rangePattern* be *rangePatterns*.[[<*fieldName*>]]; else let *rangePattern* be **undefined**.
    c. If *selectedRangePattern* is not **undefined** and *rangePattern* is **undefined**, then
       i. NOTE: Because there is no range pattern for differences at or below this field, no further checks will be performed.
       ii. Set *checkMoreFields* to **false**.
    d. If *relevantFieldsEqual* is **true** and *checkMoreFields* is **true**, then
       i. Set *selectedRangePattern* to *rangePattern*.
       ii. If *fieldName* is equal to [[AmPm]], then
          1. If *tm1*.[[Hour]] is less than 12, let *v1* be **"am"**; else let *v1* be **"pm"**.
          2. If *tm2*.[[Hour]] is less than 12, let *v2* be **"am"**; else let *v2* be **"pm"**.
       iii. Else if *fieldName* is equal to [[DayPeriod]], then
          1. Let *v1* be a String value representing the day period of *tm1*; the String value depends upon the implementation and the effective locale of *dateTimeFormat*.
          2. Let *v2* be a String value representing the day period of *tm2*; the String value depends upon the implementation and the effective locale of *dateTimeFormat*.
       iv. Else if *fieldName* is equal to [[FractionalSecondDigits]], then
          1. Let *fractionalSecondDigits* be *dateTimeFormat*.[[FractionalSecondDigits]].
          2. If *fractionalSecondDigits* is **undefined**, then
             a. Set *fractionalSecondDigits* to 3.
          3. Let *exp* be *fractionalSecondDigits* - 3.
          4. Let *v1* be floor(*tm1*.[[Millisecond]] × 10^{exp}).

            5. Let *v2* be floor(*tm2*.[[Millisecond]] × 10^{exp}).
        v. Else,
            1. Let *v1* be *tm1*.[[<*fieldName*>]].
            2. Let *v2* be *tm2*.[[<*fieldName*>]].
        vi. If *v1* is not equal to *v2*, then
            1. Set *relevantFieldsEqual* to **false**.

12. If *relevantFieldsEqual* is **true**, then
    a. Let *collapsedResult* be a new empty List.
    b. Let *pattern* be *dateTimeFormat*.[[Pattern]].
    c. Let *patternParts* be PartitionPattern(*pattern*).
    d. Let *resultParts* be ! FormatDateTimePattern(*dateTimeFormat*, *patternParts*, *x*, **undefined**).
    e. For each Record { [[Type]], [[Value]] } *r* of *resultParts*, do
        i. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: **"shared"** } to *collapsedResult*.
    f. Return *collapsedResult*.
13. Let *rangeResult* be a new empty List.
14. If *selectedRangePattern* is **undefined**, then
    a. Set *selectedRangePattern* to *rangePatterns*.[[Default]].
15. For each Record { [[Pattern]], [[Source]] } *rangePatternPart* of *selectedRangePattern*.[[PatternParts]], do
    a. Let *pattern* be *rangePatternPart*.[[Pattern]].
    b. Let *source* be *rangePatternPart*.[[Source]].
    c. If *source* is **"startRange"** or **"shared"**, then
        i. Let *z* be *x*.
    d. Else,
        i. Let *z* be *y*.
    e. Let *patternParts* be PartitionPattern(*pattern*).
    f. Let *resultParts* be ! FormatDateTimePattern(*dateTimeFormat*, *patternParts*, *z*, *selectedRangePattern*).
    g. For each Record { [[Type]], [[Value]] } *r* of *resultParts*, do
        i. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: *source* } to *rangeResult*.
16. Return *rangeResult*.

11.5.10 FormatDateTimeRange (*dateTimeFormat*, *x*, *y*)

The abstract operation FormatDateTimeRange takes arguments *dateTimeFormat* (an Intl.DateTimeFormat), *x* (a Number), and *y* (a Number) and returns either a normal completion containing a String or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionDateTimeRangePattern(*dateTimeFormat*, *x*, *y*).
2. Let *result* be the empty String.
3. For each Record { [[Type]], [[Value]], [[Source]] } *part* of *parts*, do
    a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

11.5.11 FormatDateTimeRangeToParts (*dateTimeFormat*, *x*, *y*)

The abstract operation FormatDateTimeRangeToParts takes arguments *dateTimeFormat* (an Intl.DateTime-Format), *x* (a Number), and *y* (a Number) and returns either a normal completion containing an Array or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionDateTimeRangePattern(*dateTimeFormat*, *x*, *y*).
2. Let *result* be ! ArrayCreate(0).
3. Let *n* be 0.
4. For each Record { [[Type]], [[Value]], [[Source]] } *part* of *parts*, do
    a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
    b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
    c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
    d. Perform ! CreateDataPropertyOrThrow(*O*, **"source"**, *part*.[[Source]]).
    e. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString(𝔽(*n*)), *O*).
    f. Increment *n* by 1.
5. Return *result*.

11.5.12 ToLocalTime (*epochNs*, *calendar*, *timeZoneIdentifier*)

The implementation-defined abstract operation ToLocalTime takes arguments *epochNs* (a BigInt), *calendar* (a String), and *timeZoneIdentifier* (a String) and returns a ToLocalTime Record. It performs the following steps when called:

1. If IsTimeZoneOffsetString(*timeZoneIdentifier*) is **true**, then
   a. Let *offsetNs* be ParseTimeZoneOffsetString(*timeZoneIdentifier*).
2. Else,
   a. Assert: IsValidTimeZoneName(*timeZoneIdentifier*) is **true**.
   b. Let *offsetNs* be GetNamedTimeZoneOffsetNanoseconds(*timeZoneIdentifier*, *epochNs*).
3. Let *tz* be ℝ(*epochNs*) + *offsetNs*.
4. If *calendar* is **"gregory"**, then
   a. Return a ToLocalTime Record with fields calculated from *tz* according to Table 8.
5. Else,
   a. Return a ToLocalTime Record with the fields calculated from *tz* for the given *calendar*. The calculations should use best available information about the specified *calendar*.

NOTE A conforming implementation must recognize **"UTC"** and all Zone and Link names from the IANA Time Zone Database (and **only** such names), and use best available current and historical information about their offsets from UTC and their daylight saving time rules in calculations.

11.5.13 ToLocalTime Records

Each *ToLocalTime Record* has the fields defined in Table 8.

Table 8: Record returned by ToLocalTime

Field Name	Value Type	Value Calculation for Gregorian Calendar
[[Weekday]]	an integer	ℝ(WeekDay(𝔽(floor(*tz* / 10^6))))
[[Era]]	a String	Let *year* be YearFromTime(𝔽(floor(*tz* / 10^6))). If *year* < $\mathbf{1}_{\mathbb{F}}$, return **"BC"**, else return **"AD"**.
[[Year]]	an integer	ℝ(YearFromTime(𝔽(floor(*tz* / 10^6))))
[[RelatedYear]]	an integer or **undefined**	**undefined**
[[YearName]]	a String or **undefined**	**undefined**
[[Month]]	an integer	ℝ(MonthFromTime(𝔽(floor(*tz* / 10^6))))
[[Day]]	an integer	ℝ(DateFromTime(𝔽(floor(*tz* / 10^6))))
[[Hour]]	an integer	ℝ(HourFromTime(𝔽(floor(*tz* / 10^6))))
[[Minute]]	an integer	ℝ(MinFromTime(𝔽(floor(*tz* / 10^6))))
[[Second]]	an integer	ℝ(SecFromTime(𝔽(floor(*tz* / 10^6))))
[[Millisecond]]	an integer	ℝ(msFromTime(𝔽(floor(*tz* / 10^6))))

Table 8: Record returned by ToLocalTime *(continued)*

Field Name	Value Type	Value Calculation for Gregorian Calendar
[[InDST]]	a Boolean	Calculate **true** or **false** using the best available information about the specified *calendar* and *timeZoneIdentifier*, including current and historical information from the IANA Time Zone Database about time zone offsets from UTC and daylight saving time rules.

NORMATIVE OPTIONAL

11.5.14 UnwrapDateTimeFormat (*dtf*)

The abstract operation UnwrapDateTimeFormat takes argument *dtf* (an ECMAScript language value) and returns either a normal completion containing an ECMAScript language value or a throw completion. It returns the DateTimeFormat instance of its input object, which is either the value itself or a value associated with it by %Intl.DateTimeFormat% according to the normative optional constructor mode of 4.3 Note 1. It performs the following steps when called:

1. If Type(*dtf*) is not Object, throw a **TypeError** exception.
2. If *dtf* does not have an [[InitializedDateTimeFormat]] internal slot and ? OrdinaryHasInstance(%Intl.DateTimeFormat%, *dtf*) is **true**, then
   a. Return ? Get(*dtf*, %Intl%.[[FallbackSymbol]]).
3. Return *dtf*.

12 DisplayNames Objects

12.1 The Intl.DisplayNames Constructor

The DisplayNames constructor is the *%Intl.DisplayNames%* 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.

12.1.1 Intl.DisplayNames (*locales*, *options*)

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

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *displayNames* be ? OrdinaryCreateFromConstructor(NewTarget, **"%Intl.DisplayNames.prototype%"**, « [[InitializedDisplayNames]], [[Locale]], [[Style]], [[Type]], [[Fallback]], [[LanguageDisplay]], [[Fields]] »).
3. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
4. If *options* is **undefined**, throw a **TypeError** exception.
5. Set *options* to ? GetOptionsObject(*options*).
6. Let *opt* be a new Record.
7. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, STRING, « **"lookup"**, **"best fit"** », **"best fit"**).
8. Set *opt*.[[localeMatcher]] to *matcher*.
9. Let *r* be ResolveLocale(%Intl.DisplayNames%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.DisplayNames%.[[RelevantExtensionKeys]], %Intl.DisplayNames%.[[LocaleData]]).
10. Let *style* be ? GetOption(*options*, **"style"**, STRING, « **"narrow"**, **"short"**, **"long"** », **"long"**).
11. Set *displayNames*.[[Style]] to *style*.
12. Let *type* be ? GetOption(*options*, **"type"**, STRING, « **"language"**, **"region"**, **"script"**, **"currency"**, **"calendar"**, **"dateTimeField"** », **undefined**).
13. If *type* is **undefined**, throw a **TypeError** exception.
14. Set *displayNames*.[[Type]] to *type*.
15. Let *fallback* be ? GetOption(*options*, **"fallback"**, STRING, « **"code"**, **"none"** », **"code"**).

16. Set *displayNames*.[[Fallback]] to *fallback*.
17. Set *displayNames*.[[Locale]] to *r*.[[Locale]].
18. Let *resolvedLocaleData* be *r*.[[LocaleData]].
19. Let *types* be *resolvedLocaleData*.[[types]].
20. Assert: *types* is a Record (see 12.2.3).
21. Let *languageDisplay* be ? GetOption(*options*, **"languageDisplay"**, STRING, « **"dialect"**, **"standard"** », **"dialect"**).
22. Let *typeFields* be *types*.[[<*type*>]].
23. Assert: *typeFields* is a Record (see 12.2.3).
24. If *type* is **"language"**, then
    a. Set *displayNames*.[[LanguageDisplay]] to *languageDisplay*.
    b. Set *typeFields* to *typeFields*.[[<*languageDisplay*>]].
    c. Assert: *typeFields* is a Record (see 12.2.3).
25. Let *styleFields* be *typeFields*.[[<*style*>]].
26. Assert: *styleFields* is a Record (see 12.2.3).
27. Set *displayNames*.[[Fields]] to *styleFields*.
28. Return *displayNames*.

12.2 Properties of the Intl.DisplayNames Constructor

The Intl.DisplayNames constructor has the following properties:

12.2.1 Intl.DisplayNames.prototype

The value of **`Intl.DisplayNames.prototype`** is %Intl.DisplayNames.prototype%.

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

12.2.2 Intl.DisplayNames.supportedLocalesOf (*locales* [, *options*])

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

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

12.2.3 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 « ».

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

- [[LocaleData]].[[<*locale*>]] must have a [[types]] field for all locale values *locale*. The value of this field must be a Record, which must have fields with the names of all display name types: **"language"**, **"region"**, **"script"**, **"currency"**, **"calendar"**, and **"dateTimeField"**.
- The value of the field **"language"** must be a Record which must have fields with the names of one of the valid language displays: **"dialect"** and **"standard"**.
- The language display fields under display name type **"language"** should contain Records which must have fields with the names of one of the valid display name styles: **"narrow"**, **"short"**, and **"long"**.
- The value of the fields **"region"**, **"script"**, **"currency"**, **"calendar"**, and **"dateTimeField"** must be Records, which must have fields with the names of all display name styles: **"narrow"**, **"short"**, and **"long"**.
- The display name style fields under display name type **"language"** should contain Records with keys corresponding to language codes that can be matched by the **`unicode_language_id`** Unicode locale nonterminal. The value of these fields must be string values.

- The display name style fields under display name type **"region"** should contain Records with keys corresponding to region codes. The value of these fields must be string values.
- The display name style fields under display name type **"script"** should contain Records with keys corresponding to script codes. The value of these fields must be string values.
- The display name style fields under display name type **"currency"** should contain Records with keys corresponding to currency codes. The value of these fields must be string values.
- The display name style fields under display name type **"calendar"** should contain Records with keys corresponding to calendar identifiers that can be matched by the `type` Unicode locale nonterminal. The value of these fields must be string values.
- The display name style fields under display name type **"dateTimeField"** should contain Records with keys corresponding to codes listed in Table `10`. The value of these fields must be string values.

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

12.3 Properties of the Intl.DisplayNames Prototype Object

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

12.3.1 Intl.DisplayNames.prototype.constructor

The initial value of `Intl.DisplayNames.prototype.constructor` is %Intl.DisplayNames%.

12.3.2 Intl.DisplayNames.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.DisplayNames"**.

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

12.3.3 Intl.DisplayNames.prototype.of (*code*)

When the `Intl.DisplayNames.prototype.of` is called with an argument *code*, the following steps are taken:

1. Let *displayNames* be **this** value.
2. Perform ? RequireInternalSlot(*displayNames*, [[InitializedDisplayNames]]).
3. Let *code* be ? ToString(*code*).
4. Set *code* to ? CanonicalCodeForDisplayNames(*displayNames*.[[Type]], *code*).
5. Let *fields* be *displayNames*.[[Fields]].
6. If *fields* has a field [[<*code*>]], return *fields*.[[<*code*>]].
7. If *displayNames*.[[Fallback]] is **"code"**, return *code*.
8. Return **undefined**.

12.3.4 Intl.DisplayNames.prototype.resolvedOptions ( )

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

1. Let *displayNames* be **this** value.
2. Perform ? RequireInternalSlot(*displayNames*, [[InitializedDisplayNames]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 9, 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 *displayNames*'s internal slot whose name is the Internal Slot value of the current row.
   c. Assert: *v* is not **undefined**.

   d. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
5. Return *options*.

Table 9: Resolved Options of DisplayNames Instances

Internal Slot	Property
[[Locale]]	**"locale"**
[[Style]]	**"style"**
[[Type]]	**"type"**
[[Fallback]]	**"fallback"**
[[LanguageDisplay]]	**"languageDisplay"**

12.4 Properties of Intl.DisplayNames Instances

Intl.DisplayNames instances are ordinary objects that inherit properties from %Intl.DisplayNames.prototype%.

Intl.DisplayNames instances have an [[InitializedDisplayNames]] internal slot.

Intl.DisplayNames 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.
- [[Style]] is one of the String values **"narrow"**, **"short"**, or **"long"**, identifying the display name style used.
- [[Type]] is one of the String values **"language"**, **"region"**, **"script"**, **"currency"**, **"calendar"**, or **"dateTimeField"**, identifying the type of the display names requested.
- [[Fallback]] is one of the String values **"code"** or **"none"**, identifying the fallback return when the system does not have the requested display name.
- [[LanguageDisplay]] is one of the String values **"dialect"** or **"standard"**, identifying the language display kind. It is only used when [[Type]] has the value **"language"**.
- [[Fields]] is a Record (see 12.2.3) which must have fields with keys corresponding to codes according to [[Style]], [[Type]], and [[LanguageDisplay]].

12.5 Abstract Operations for DisplayNames Objects

12.5.1 CanonicalCodeForDisplayNames (*type*, *code*)

The abstract operation CanonicalCodeForDisplayNames takes arguments *type* (a String) and *code* (a String) and returns either a normal completion containing a String or a throw completion. It verifies that *code* represents a well-formed code according to *type* and returns the case-regularized form of *code*. It performs the following steps when called:

1. If *type* is **"language"**, then
   a. If *code* cannot be matched by the **`unicode_language_id`** Unicode locale nonterminal, throw a **RangeError** exception.
   b. If IsStructurallyValidLanguageTag(*code*) is **false**, throw a **RangeError** exception.
   c. Return CanonicalizeUnicodeLocaleId(*code*).
2. If *type* is **"region"**, then
   a. If *code* cannot be matched by the **`unicode_region_subtag`** Unicode locale nonterminal, throw a **RangeError** exception.
   b. Return the ASCII-uppercase of *code*.
3. If *type* is **"script"**, then
   a. If *code* cannot be matched by the **`unicode_script_subtag`** Unicode locale nonterminal, throw a **RangeError** exception.
   b. Assert: The length of *code* is 4, and every code unit of *code* represents an ASCII letter (`0x0041` through `0x005A` and `0x0061` through `0x007A`, both inclusive).

   c. Let *first* be the ASCII-uppercase of the substring of *code* from 0 to 1.
   d. Let *rest* be the ASCII-lowercase of the substring of *code* from 1.
   e. Return the string-concatenation of *first* and *rest*.
4. If *type* is **"calendar"**, then
   a. If *code* cannot be matched by the **type** Unicode locale nonterminal, throw a **RangeError** exception.
   b. If *code* uses any of the backwards compatibility syntax described in Unicode Technical Standard #35 Part 1 Core, Section 3.3 BCP 47 Conformance <https://unicode.org/reports/tr35/#BCP_47_Conformance>, throw a **RangeError** exception.
   c. Return the ASCII-lowercase of *code*.
5. If *type* is **"dateTimeField"**, then
   a. If the result of IsValidDateTimeFieldCode(*code*) is **false**, throw a **RangeError** exception.
   b. Return *code*.
6. Assert: *type* is **"currency"**.
7. If IsWellFormedCurrencyCode(*code*) is **false**, throw a **RangeError** exception.
8. Return the ASCII-uppercase of *code*.

12.5.2 IsValidDateTimeFieldCode (*field*)

The abstract operation IsValidDateTimeFieldCode takes argument *field* (a String) and returns a Boolean. It verifies that the *field* argument represents a valid date time field code. It performs the following steps when called:

1. If *field* is listed in the Code column of Table 10, return **true**.
2. Return **false**.

Table 10: Codes For Date Time Field of DisplayNames

Code	Description
"era"	The field indicating the era, e.g. AD or BC in the Gregorian or Julian calendar.
"year"	The field indicating the year (within an era).
"quarter"	The field indicating the quarter, e.g. Q2, 2nd quarter, etc.
"month"	The field indicating the month, e.g. Sep, September, etc.
"weekOfYear"	The field indicating the week number within a year.
"weekday"	The field indicating the day of week, e.g. Tue, Tuesday, etc.
"day"	The field indicating the day in month.
"dayPeriod"	The field indicating the day period, either am, pm, etc. or noon, evening, etc..
"hour"	The field indicating the hour.
"minute"	The field indicating the minute.
"second"	The field indicating the second.
"timeZoneName"	The field indicating the time zone name, e.g. PDT, Pacific Daylight Time, etc.

13 ListFormat Objects

13.1 The Intl.ListFormat Constructor

The ListFormat constructor is the *%Intl.ListFormat%* 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.

13.1.1 Intl.ListFormat ([*locales* [, *options*]])

When the **`Intl.ListFormat`** 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 *listFormat* be ? OrdinaryCreateFromConstructor(NewTarget, **"%Intl.ListFormat.prototype%"**, « [[InitializedListFormat]], [[Locale]], [[Type]], [[Style]], [[Templates]] »).
3. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
4. Set *options* to ? GetOptionsObject(*options*).
5. Let *opt* be a new Record.
6. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, STRING, « **"lookup"**, **"best fit"** », **"best fit"**).
7. Set *opt*.[[localeMatcher]] to *matcher*.
8. Let *r* be ResolveLocale(%Intl.ListFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.ListFormat%.[[RelevantExtensionKeys]], %Intl.ListFormat%.[[LocaleData]]).
9. Set *listFormat*.[[Locale]] to *r*.[[Locale]].
10. Let *type* be ? GetOption(*options*, **"type"**, STRING, « **"conjunction"**, **"disjunction"**, **"unit"** », **"conjunction"**).
11. Set *listFormat*.[[Type]] to *type*.
12. Let *style* be ? GetOption(*options*, **"style"**, STRING, « **"long"**, **"short"**, **"narrow"** », **"long"**).
13. Set *listFormat*.[[Style]] to *style*.
14. Let *resolvedLocaleData* be *r*.[[LocaleData]].
15. Let *dataLocaleTypes* be *resolvedLocaleData*.[[<*type*>]].
16. Set *listFormat*.[[Templates]] to *dataLocaleTypes*.[[<*style*>]].
17. Return *listFormat*.

13.2 Properties of the Intl.ListFormat Constructor

The Intl.ListFormat constructor has the following properties:

13.2.1 Intl.ListFormat.prototype

The value of **`Intl.ListFormat.prototype`** is %Intl.ListFormat.prototype%.

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

13.2.2 Intl.ListFormat.supportedLocalesOf (*locales* [, *options*])

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

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

13.2.3 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 « ».

NOTE 1 Intl.ListFormat does not have any relevant extension keys.

The value of the [[LocaleData]] internal slot is implementation-defined within the constraints described in 9.1 and the following additional constraints, for each locale value *locale* in %Intl.ListFormat%.[[AvailableLocales]]:

- [[LocaleData]].[[<*locale*>]] is a Record which has three fields [[conjunction]], [[disjunction]], and [[unit]]. Each of these is a Record which must have fields with the names of three formatting styles: [[long]], [[short]], and [[narrow]].

- Each of those fields is considered a *ListFormat template set*, which must be a List of Records with fields named: [[Pair]], [[Start]], [[Middle]], and [[End]]. Each of those fields must be a template string as specified in LDML List Format Rules. Each template string must contain the substrings **"{0}"** and **"{1}"** exactly once. The substring **"{0}"** should occur before the substring **"{1}"**.

NOTE 2 It is recommended that implementations use the locale data provided by the Common Locale Data Repository (available at https://cldr.unicode.org/). In LDML's listPattern <https://unicode.org/reports/tr35/tr35-general.html#ListPatterns>, **`conjunction`** corresponds to "standard", **`disjunction`** corresponds to "or", and **`unit`** corresponds to "unit".

NOTE 3 Among the list types, **`conjunction`** stands for "and"-based lists (e.g., "A, B, and C"), **`disjunction`** stands for "or"-based lists (e.g., "A, B, or C"), and **`unit`** stands for lists of values with units (e.g., "5 pounds, 12 ounces").

13.3 Properties of the Intl.ListFormat Prototype Object

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

13.3.1 Intl.ListFormat.prototype.constructor

The initial value of **`Intl.ListFormat.prototype.constructor`** is %Intl.ListFormat%.

13.3.2 Intl.ListFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.ListFormat"**.

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

13.3.3 Intl.ListFormat.prototype.format (*list*)

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

1. Let *lf* be the **this** value.
2. Perform ? RequireInternalSlot(*lf*, [[InitializedListFormat]]).
3. Let *stringList* be ? StringListFromIterable(*list*).
4. Return FormatList(*lf*, *stringList*).

13.3.4 Intl.ListFormat.prototype.formatToParts (*list*)

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

1. Let *lf* be the **this** value.
2. Perform ? RequireInternalSlot(*lf*, [[InitializedListFormat]]).
3. Let *stringList* be ? StringListFromIterable(*list*).
4. Return FormatListToParts(*lf*, *stringList*).

13.3.5 Intl.ListFormat.prototype.resolvedOptions ( )

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

1. Let *lf* be the **this** value.
2. Perform ? RequireInternalSlot(*lf*, [[InitializedListFormat]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 11, 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 *lf*'s internal slot whose name is the Internal Slot value of the current row.
   c. Assert: *v* is not **undefined**.
   d. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
5. Return *options*.

Table 11: Resolved Options of ListFormat Instances

Internal Slot	Property
[[Locale]]	**"locale"**
[[Type]]	**"type"**
[[Style]]	**"style"**

13.4 Properties of Intl.ListFormat Instances

Intl.ListFormat instances inherit properties from %Intl.ListFormat.prototype%.

Intl.ListFormat instances have an [[InitializedListFormat]] internal slot.

Intl.ListFormat 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 list format styles.
- [[Type]] is one of the String values **"conjunction"**, **"disjunction"**, or **"unit"**, identifying the list of types used.
- [[Style]] is one of the String values **"long"**, **"short"**, or **"narrow"**, identifying the list formatting style used.
- [[Templates]] is a ListFormat template set.

13.5 Abstract Operations for ListFormat Objects

13.5.1 DeconstructPattern (*pattern*, *placeables*)

The abstract operation DeconstructPattern takes arguments *pattern* (a String) and *placeables* (a Record) and returns a List.

It deconstructs the pattern string into a List of parts.

placeables is a Record whose keys are placeables tokens used in the pattern string, and values are parts records (as from PartitionPattern) which will be used in the result List to represent the token part. Example:

```
Input:
  DeconstructPattern("AA{xx}BB{yy}CC", {
    [[xx]]: {[[Type]]: "hour", [[Value]]: "15"},
    [[yy]]: {[[Type]]: "minute", [[Value]]: "06"}
  })

Output (List of parts records):
  «
    {[[Type]]: "literal", [[Value]]: "AA"},
    {[[Type]]: "hour", [[Value]]: "15"},
    {[[Type]]: "literal", [[Value]]: "BB"},
    {[[Type]]: "minute", [[Value]]: "06"},
    {[[Type]]: "literal", [[Value]]: "CC"}
  »
```

It performs the following steps when called:

1. Let *patternParts* be PartitionPattern(*pattern*).
2. Let *result* be a new empty List.
3. For each Record { [[Type]], [[Value]] } *patternPart* of *patternParts*, do
    a. Let *part* be *patternPart*.[[Type]].
    b. If *part* is **"literal"**, then
        i. Append the Record { [[Type]]: **"literal"**, [[Value]]: *patternPart*.[[Value]] } to *result*.
    c. Else,
        i. Assert: *placeables* has a field [[<*part*>]].
        ii. Let *subst* be *placeables*.[[<*part*>]].
        iii. If Type(*subst*) is List, then
            1. For each element *s* of *subst*, do
                a. Append *s* to *result*.
        iv. Else,
            1. Append *subst* to *result*.
4. Return *result*.

13.5.2 CreatePartsFromList (*listFormat*, *list*)

The abstract operation CreatePartsFromList takes arguments *listFormat* (an Intl.ListFormat) and *list* (a List of Strings) and returns a List of Records with fields [[Type]] (**"element"** or **"literal"**) and [[Value]] (a String). It creates the corresponding list of parts according to the effective locale and the formatting options of *listFormat*. It performs the following steps when called:

1. Let *size* be the number of elements of *list*.
2. If *size* is `0`, then
    a. Return a new empty List.
3. If *size* is 2, then
    a. Let *n* be an index into *listFormat*.[[Templates]] based on *listFormat*.[[Locale]], *list*[`0`], and *list*[1].
    b. Let *pattern* be *listFormat*.[[Templates]][*n*].[[Pair]].
    c. Let *first* be a new Record { [[Type]]: **"element"**, [[Value]]: *list*[`0`] }.
    d. Let *second* be a new Record { [[Type]]: **"element"**, [[Value]]: *list*[1] }.
    e. Let *placeables* be a new Record { [[`0`]]: *first*, [[1]]: *second* }.
    f. Return DeconstructPattern(*pattern*, *placeables*).
4. Let *last* be a new Record { [[Type]]: **"element"**, [[Value]]: *list*[*size* - 1] }.
5. Let *parts* be « *last* ».
6. Let *i* be *size* - 2.
7. Repeat, while *i* ≥ `0`,
    a. Let *head* be a new Record { [[Type]]: **"element"**, [[Value]]: *list*[*i*] }.
    b. Let *n* be an implementation-defined index into *listFormat*.[[Templates]] based on *listFormat*.[[Locale]], *head*, and *parts*.
    c. If *i* is `0`, then
        i. Let *pattern* be *listFormat*.[[Templates]][*n*].[[Start]].

   d. Else if *i* is less than *size* - 2, then
      i. Let *pattern* be *listFormat*.[[Templates]][*n*].[[Middle]].
   e. Else,
      i. Let *pattern* be *listFormat*.[[Templates]][*n*].[[End]].
   f. Let *placeables* be a new Record { [[0]]: *head*, [[1]]: *parts* }.
   g. Set *parts* to DeconstructPattern(*pattern*, *placeables*).
   h. Decrement *i* by 1.
8. Return *parts*.

NOTE The index *n* to select across multiple templates permits the conjunction to be dependent on the context, as in Spanish, where either "y" or "e" may be selected, depending on the following word.

13.5.3 FormatList (*listFormat*, *list*)

The abstract operation FormatList takes arguments *listFormat* (an Intl.ListFormat) and *list* (a List of Strings) and returns a String. It performs the following steps when called:

1. Let *parts* be CreatePartsFromList(*listFormat*, *list*).
2. Let *result* be an empty String.
3. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

13.5.4 FormatListToParts (*listFormat*, *list*)

The abstract operation FormatListToParts takes arguments *listFormat* (an Intl.ListFormat) and *list* (a List of Strings) and returns an Array. It performs the following steps when called:

1. Let *parts* be CreatePartsFromList(*listFormat*, *list*).
2. Let *result* be ! ArrayCreate(0).
3. Let *n* be 0.
4. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
   d. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString($\mathbb{F}$(*n*)), *O*).
   e. Increment *n* by 1.
5. Return *result*.

13.5.5 StringListFromIterable (*iterable*)

The abstract operation StringListFromIterable takes argument *iterable* (an ECMAScript language value) and returns either a normal completion containing a List of Strings or a throw completion. It performs the following steps when called:

1. If *iterable* is **undefined**, then
   a. Return a new empty List.
2. Let *iteratorRecord* be ? GetIterator(*iterable*, SYNC).
3. Let *list* be a new empty List.
4. Repeat,
   a. Let *next* be ? IteratorStepValue(*iteratorRecord*).
   b. If *next* is DONE, then
      i. Return *list*.
   c. If Type(*next*) is not String, then
      i. Let *error* be ThrowCompletion(a newly created **TypeError** object).
      ii. Return ? IteratorClose(*iteratorRecord*, *error*).
   d. Append *next* to *list*.

NOTE This algorithm raises exceptions when it encounters values that are not Strings, because there is no obvious locale-aware coercion for arbitrary values.

14 Locale Objects

14.1 The Intl.Locale Constructor

The Locale constructor is the *%Intl.Locale%* intrinsic object and a standard built-in property of the Intl object.

14.1.1 Intl.Locale (*tag* [, *options*])

When the `Intl.Locale` function is called with an argument *tag* and an optional argument *options*, the following steps are taken:

1. If NewTarget is **undefined**, throw a **TypeError** exception.
2. Let *relevantExtensionKeys* be %Intl.Locale%.[[RelevantExtensionKeys]].
3. Let *internalSlotsList* be « [[InitializedLocale]], [[Locale]], [[Calendar]], [[Collation]], [[HourCycle]], [[NumberingSystem]] ».
4. If *relevantExtensionKeys* contains **"kf"**, then
   a. Append [[CaseFirst]] to *internalSlotsList*.
5. If *relevantExtensionKeys* contains **"kn"**, then
   a. Append [[Numeric]] to *internalSlotsList*.
6. Let *locale* be ? OrdinaryCreateFromConstructor(NewTarget, **"%Intl.Locale.prototype%"**, *internalSlotsList*).
7. If Type(*tag*) is not String or Object, throw a **TypeError** exception.
8. If Type(*tag*) is Object and *tag* has an [[InitializedLocale]] internal slot, then
   a. Let *tag* be *tag*.[[Locale]].
9. Else,
   a. Let *tag* be ? ToString(*tag*).
10. Set *options* to ? CoerceOptionsToObject(*options*).
11. Set *tag* to ? ApplyOptionsToTag(*tag*, *options*).
12. Let *opt* be a new Record.
13. Let *calendar* be ? GetOption(*options*, **"calendar"**, STRING, EMPTY, **undefined**).
14. If *calendar* is not **undefined**, then
    a. If *calendar* cannot be matched by the `type` Unicode locale nonterminal, throw a **RangeError** exception.
15. Set *opt*.[[ca]] to *calendar*.
16. Let *collation* be ? GetOption(*options*, **"collation"**, STRING, EMPTY, **undefined**).
17. If *collation* is not **undefined**, then
    a. If *collation* cannot be matched by the `type` Unicode locale nonterminal, throw a **RangeError** exception.
18. Set *opt*.[[co]] to *collation*.
19. Let *hc* be ? GetOption(*options*, **"hourCycle"**, STRING, « **"h11"**, **"h12"**, **"h23"**, **"h24"** », **undefined**).
20. Set *opt*.[[hc]] to *hc*.
21. Let *kf* be ? GetOption(*options*, **"caseFirst"**, STRING, « **"upper"**, **"lower"**, **"false"** », **undefined**).
22. Set *opt*.[[kf]] to *kf*.
23. Let *kn* be ? GetOption(*options*, **"numeric"**, BOOLEAN, EMPTY, **undefined**).
24. If *kn* is not **undefined**, set *kn* to ! ToString(*kn*).
25. Set *opt*.[[kn]] to *kn*.
26. Let *numberingSystem* be ? GetOption(*options*, **"numberingSystem"**, STRING, EMPTY, **undefined**).
27. If *numberingSystem* is not **undefined**, then
    a. If *numberingSystem* cannot be matched by the `type` Unicode locale nonterminal, throw a **RangeError** exception.
28. Set *opt*.[[nu]] to *numberingSystem*.
29. Let *r* be ApplyUnicodeExtensionToTag(*tag*, *opt*, *relevantExtensionKeys*).
30. Set *locale*.[[Locale]] to *r*.[[locale]].
31. Set *locale*.[[Calendar]] to *r*.[[ca]].
32. Set *locale*.[[Collation]] to *r*.[[co]].
33. Set *locale*.[[HourCycle]] to *r*.[[hc]].
34. If *relevantExtensionKeys* contains **"kf"**, then

   a. Set *locale*.[[CaseFirst]] to *r*.[[kf]].
35. If *relevantExtensionKeys* contains **"kn"**, then
   a. If SameValue(*r*.[[kn]], **"true"**) is **true** or *r*.[[kn]] is the empty String, then
      i. Set *locale*.[[Numeric]] to **true**.
   b. Else,
      i. Set *locale*.[[Numeric]] to **false**.
36. Set *locale*.[[NumberingSystem]] to *r*.[[nu]].
37. Return *locale*.

14.1.2 ApplyOptionsToTag (*tag*, *options*)

The abstract operation ApplyOptionsToTag takes arguments *tag* (a String) and *options* (an Object) and returns either a normal completion containing a Unicode canonicalized locale identifier or a throw completion. It performs the following steps when called:

1. If IsStructurallyValidLanguageTag(*tag*) is **false**, throw a **RangeError** exception.
2. Let *language* be ? GetOption(*options*, **"language"**, STRING, EMPTY, **undefined**).
3. If *language* is not **undefined**, then
   a. If *language* cannot be matched by the **`unicode_language_subtag`** Unicode locale nonterminal, throw a **RangeError** exception.
4. Let *script* be ? GetOption(*options*, **"script"**, STRING, EMPTY, **undefined**).
5. If *script* is not **undefined**, then
   a. If *script* cannot be matched by the **`unicode_script_subtag`** Unicode locale nonterminal, throw a **RangeError** exception.
6. Let *region* be ? GetOption(*options*, **"region"**, STRING, EMPTY, **undefined**).
7. If *region* is not **undefined**, then
   a. If *region* cannot be matched by the **`unicode_region_subtag`** Unicode locale nonterminal, throw a **RangeError** exception.
8. Set *tag* to CanonicalizeUnicodeLocaleId(*tag*).
9. Assert: *tag* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
10. Let *languageId* be the longest prefix of *tag* matched by the **`unicode_language_id`** Unicode locale nonterminal.
11. If *language* is **undefined**, set *language* to GetLocaleLanguage(*languageId*).
12. If *script* is **undefined**, set *script* to GetLocaleScript(*languageId*).
13. If *region* is **undefined**, set *region* to GetLocaleRegion(*languageId*).
14. Let *variants* be GetLocaleVariants(*languageId*).
15. Set *languageId* to *language*.
16. If *script* is not **undefined**, set *languageId* to the string-concatenation of *languageId*, **"-"**, and *script*.
17. If *region* is not **undefined**, set *languageId* to the string-concatenation of *languageId*, **"-"**, and *region*.
18. If *variants* is not **undefined**, set *languageId* to the string-concatenation of *languageId*, **"-"**, and *variants*.
19. Set *tag* to *tag* with the substring matched by the **`unicode_language_id`** Unicode locale nonterminal replaced by the string *languageId*.
20. Return CanonicalizeUnicodeLocaleId(*tag*).

14.1.3 ApplyUnicodeExtensionToTag (*tag*, *options*, *relevantExtensionKeys*)

The abstract operation ApplyUnicodeExtensionToTag takes arguments *tag* (a String), *options* (a Record), and *relevantExtensionKeys* (a List of Strings) and returns a Record. It performs the following steps when called:

1. Assert: *tag* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
2. If *tag* contains a substring that is a Unicode locale extension sequence, then
   a. Let *extension* be the String value consisting of the substring of the Unicode locale extension sequence within *tag*.
   b. Let *components* be UnicodeExtensionComponents(*extension*).
   c. Let *attributes* be *components*.[[Attributes]].
   d. Let *keywords* be *components*.[[Keywords]].
3. Else,
   a. Let *attributes* be a new empty List.
   b. Let *keywords* be a new empty List.
4. Let *result* be a new Record.

5. For each element *key* of *relevantExtensionKeys*, do
   a. Let *value* be **undefined**.
   b. If *keywords* contains an element whose [[Key]] is the same as *key*, then
      i. Let *entry* be the element of *keywords* whose [[Key]] is the same as *key*.
      ii. Set *value* to *entry*.[[Value]].
   c. Else,
      i. Let *entry* be EMPTY.
   d. Assert: *options* has a field [[<*key*>]].
   e. Let *optionsValue* be *options*.[[<*key*>]].
   f. If *optionsValue* is not **undefined**, then
      i. Assert: Type(*optionsValue*) is String.
      ii. Set *value* to *optionsValue*.
      iii. If *entry* is not EMPTY, then
         1. Set *entry*.[[Value]] to *value*.
      iv. Else,
         1. Append the Record { [[Key]]: *key*, [[Value]]: *value* } to *keywords*.
   g. Set *result*.[[<*key*>]] to *value*.
6. Let *locale* be the String value that is *tag* with any Unicode locale extension sequences removed.
7. Let *newExtension* be a Unicode BCP 47 U Extension based on *attributes* and *keywords*.
8. If *newExtension* is not the empty String, then
   a. Set *locale* to InsertUnicodeExtensionAndCanonicalize(*locale*, *newExtension*).
9. Set *result*.[[locale]] to *locale*.
10. Return *result*.

14.2 Properties of the Intl.Locale Constructor

The Intl.Locale constructor has the following properties:

14.2.1 Intl.Locale.prototype

The value of `Intl.Locale.prototype` is %Intl.Locale.prototype%.

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

14.2.2 Internal slots

The value of the [[RelevantExtensionKeys]] internal slot is « **"ca"**, **"co"**, **"hc"**, **"kf"**, **"kn"**, **"nu"** ». If %Intl.Collator%.[[RelevantExtensionKeys]] does not contain **"kf"**, then remove **"kf"** from %Intl.Locale%.[[RelevantExtensionKeys]]. If %Intl.Collator%.[[RelevantExtensionKeys]] does not contain **"kn"**, then remove **"kn"** from %Intl.Locale%.[[RelevantExtensionKeys]].

14.3 Properties of the Intl.Locale Prototype Object

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

14.3.1 Intl.Locale.prototype.constructor

The initial value of `Intl.Locale.prototype.constructor` is %Intl.Locale%.

14.3.2 Intl.Locale.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.Locale"**.

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

14.3.3 Intl.Locale.prototype.maximize ( )

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Let *maximal* be the result of the Add Likely Subtags <https://unicode.org/reports/tr35/#Likely_Subtags> algorithm applied to *loc*.[[Locale]]. If an error is signaled, set *maximal* to *loc*.[[Locale]].
4. Return ! Construct(%Intl.Locale%, *maximal*).

14.3.4 Intl.Locale.prototype.minimize ( )

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Let *minimal* be the result of the Remove Likely Subtags <https://unicode.org/reports/tr35/#Likely_Subtags> algorithm applied to *loc*.[[Locale]]. If an error is signaled, set *minimal* to *loc*.[[Locale]].
4. Return ! Construct(%Intl.Locale%, *minimal*).

14.3.5 Intl.Locale.prototype.toString ( )

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[Locale]].

14.3.6 get Intl.Locale.prototype.baseName

`Intl.Locale.prototype.baseName` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Let *locale* be *loc*.[[Locale]].
4. Return the longest prefix of *locale* matched by the **`unicode_language_id`** Unicode locale nonterminal.

14.3.7 get Intl.Locale.prototype.calendar

`Intl.Locale.prototype.calendar` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[Calendar]].

14.3.8 get Intl.Locale.prototype.caseFirst

This property only exists if %Intl.Locale%.[[RelevantExtensionKeys]] contains **"kf"**.

`Intl.Locale.prototype.caseFirst` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[CaseFirst]].

14.3.9 get Intl.Locale.prototype.collation

`Intl.Locale.prototype.collation` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[Collation]].

14.3.10 get Intl.Locale.prototype.hourCycle

`Intl.Locale.prototype.hourCycle` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[HourCycle]].

14.3.11 get Intl.Locale.prototype.numeric

This property only exists if %Intl.Locale%.[[RelevantExtensionKeys]] contains **"kn"**.

`Intl.Locale.prototype.numeric` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[Numeric]].

14.3.12 get Intl.Locale.prototype.numberingSystem

`Intl.Locale.prototype.numberingSystem` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return *loc*.[[NumberingSystem]].

14.3.13 get Intl.Locale.prototype.language

`Intl.Locale.prototype.language` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return GetLocaleLanguage(*loc*.[[Locale]]).

14.3.14 get Intl.Locale.prototype.script

`Intl.Locale.prototype.script` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return GetLocaleScript(*loc*.[[Locale]]).

14.3.15 get Intl.Locale.prototype.region

`Intl.Locale.prototype.region` is an accessor property whose set accessor function is **undefined**. Its get accessor function performs the following steps:

1. Let *loc* be the **this** value.
2. Perform ? RequireInternalSlot(*loc*, [[InitializedLocale]]).
3. Return GetLocaleRegion(*loc*.[[Locale]]).

14.4 Properties of Intl.Locale Instances

Intl.Locale instances are ordinary objects that inherit properties from %Intl.Locale.prototype%.

Intl.Locale instances have an [[InitializedLocale]] internal slot.

Intl.Locale 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 that is a syntactically valid type value as given in Unicode Technical Standard #35 Part 1 Core, Section 3.2 Unicode Locale Identifier <https://unicode.org/reports/tr35/#Unicode_locale_identifier>, or is **undefined**.
- [[Collation]] is a String value that is a syntactically valid type value as given in Unicode Technical Standard #35 Part 1 Core, Section 3.2 Unicode Locale Identifier <https://unicode.org/reports/tr35/#Unicode_locale_identifier>, or is **undefined**.
- [[HourCycle]] is a String value that is a syntactically valid type value as given in Unicode Technical Standard #35 Part 1 Core, Section 3.2 Unicode Locale Identifier <https://unicode.org/reports/tr35/#Unicode_locale_identifier>, or is **undefined**.
- [[NumberingSystem]] is a String value that is a syntactically valid type value as given in Unicode Technical Standard #35 Part 1 Core, Section 3.2 Unicode Locale Identifier <https://unicode.org/reports/tr35/#Unicode_locale_identifier>, or is **undefined**.
- [[CaseFirst]] is a String value that is a syntactically valid type value as given in Unicode Technical Standard #35 Part 1 Core, Section 3.2 Unicode Locale Identifier <https://unicode.org/reports/tr35/#Unicode_locale_identifier>, or is **undefined**. This internal slot only exists if the [[RelevantExtensionKeys]] internal slot of %Intl.Locale% contains **"kf"**.
- [[Numeric]] is a Boolean value specifying whether numeric sorting is used by the locale, or is **undefined**. This internal slot only exists if the [[RelevantExtensionKeys]] internal slot of %Intl.Locale% contains **"kn"**.

14.5 Abstract Operations for Locale Objects

14.5.1 GetLocaleLanguage (*locale*)

The abstract operation GetLocaleLanguage takes argument *locale* (a String) and returns a String. It performs the following steps when called:

1. Assert: *locale* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
2. Let *languageId* be the longest prefix of *locale* matched by the **`unicode_language_id`** Unicode locale nonterminal.
3. Assert: The first subtag of *languageId* can be matched by the **`unicode_language_subtag`** Unicode locale nonterminal.
4. Return the first subtag of *languageId*.

14.5.2 GetLocaleScript (*locale*)

The abstract operation GetLocaleScript takes argument *locale* (a String) and returns a String or **undefined**. It performs the following steps when called:

1. Assert: *locale* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
2. Let *languageId* be the longest prefix of *locale* matched by the **`unicode_language_id`** Unicode locale nonterminal.
3. Assert: *languageId* contains at most one subtag that can be matched by the **`unicode_script_subtag`** Unicode locale nonterminal.
4. If *languageId* contains a subtag matched by the **`unicode_script_subtag`** Unicode locale nonterminal, return that subtag.
5. Return **undefined**.

14.5.3 GetLocaleRegion (*locale*)

The abstract operation GetLocaleRegion takes argument *locale* (a String) and returns a String or **undefined**. It performs the following steps when called:

1. Assert: *locale* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
2. Let *languageId* be the longest prefix of *locale* matched by the **`unicode_language_id`** Unicode locale nonterminal.
3. NOTE: A **`unicode_region_subtag`** subtag is only valid immediately after an initial **`unicode_language_subtag`** subtag, optionally with a single **`unicode_script_subtag`** subtag between them. In that position, **`unicode_region_subtag`** cannot be confused with any other valid subtag because all their productions are disjoint.
4. Assert: The first subtag of *languageId* can be matched by the **`unicode_language_subtag`** Unicode locale nonterminal.
5. Let *languageIdTail* be the suffix of *languageId* following the first subtag.
6. Assert: *languageIdTail* contains at most one subtag that can be matched by the **`unicode_region_subtag`** Unicode locale nonterminal.
7. If *languageIdTail* contains a subtag matched by the **`unicode_region_subtag`** Unicode locale nonterminal, return that subtag.
8. Return **undefined**.

14.5.4 GetLocaleVariants (*locale*)

The abstract operation GetLocaleVariants takes argument *locale* (a String) and returns a String or **undefined**. It performs the following steps when called:

1. Assert: *locale* can be matched by the **`unicode_locale_id`** Unicode locale nonterminal.
2. Let *languageId* be the longest prefix of *locale* matched by the **`unicode_language_id`** Unicode locale nonterminal.
3. If there is a non-empty suffix of *languageId* that is a consecutive sequence of substrings in which each element is a **"-"** followed by a substring that is matched by the **`unicode_variant_subtag`** Unicode locale nonterminal, then
   a. Let *variants* be the longest such suffix.
   b. Return the substring of *variants* from 1.
4. Return **undefined**.

15 NumberFormat Objects

15.1 The Intl.NumberFormat Constructor

The NumberFormat constructor is the *%Intl.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.

15.1.1 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*, **"%Intl.NumberFormat.prototype%"**, « [[InitializedNumberFormat]], [[Locale]], [[LocaleData]], [[NumberingSystem]], [[Style]], [[Unit]], [[UnitDisplay]], [[Currency]], [[CurrencyDisplay]], [[CurrencySign]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[RoundingType]], [[Notation]], [[CompactDisplay]], [[UseGrouping]], [[SignDisplay]], [[RoundingIncrement]], [[RoundingMode]], [[ComputedRoundingPriority]], [[TrailingZeroDisplay]], [[BoundFormat]] »).
3. Perform ? InitializeNumberFormat(*numberFormat*, *locales*, *options*).
4. If the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Let *this* be the **this** value.
   b. Return ? ChainNumberFormat(*numberFormat*, NewTarget, *this*).
5. Return *numberFormat*.

NORMATIVE OPTIONAL

15.1.1.1 ChainNumberFormat (*numberFormat*, *newTarget*, *this*)

The abstract operation ChainNumberFormat takes arguments *numberFormat* (an Intl.NumberFormat), *newTarget* (an ECMAScript language value), and *this* (an ECMAScript language value) and returns either a normal completion containing an Object or a throw completion. It performs the following steps when called:

1. If *newTarget* is **undefined** and ? OrdinaryHasInstance(%Intl.NumberFormat%, *this*) is **true**, then
   a. Perform ? DefinePropertyOrThrow(*this*, %Intl%.[[FallbackSymbol]], PropertyDescriptor{ [[Value]]: *numberFormat*, [[Writable]]: **false**, [[Enumerable]]: **false**, [[Configurable]]: **false** }).
   b. Return *this*.
2. Return *numberFormat*.

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

The abstract operation InitializeNumberFormat takes arguments *numberFormat* (an Intl.NumberFormat), *locales* (an ECMAScript language value), and *options* (an ECMAScript language value) and returns either a normal completion containing UNUSED or a throw completion. It initializes *numberFormat* as a NumberFormat object. It performs the following steps when called:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. Set *options* to ? CoerceOptionsToObject(*options*).
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 *numberingSystem* be ? GetOption(*options*, **"numberingSystem"**, STRING, EMPTY, **undefined**).
7. If *numberingSystem* is not **undefined**, then
   a. If *numberingSystem* cannot be matched by the **type** Unicode locale nonterminal, throw a **RangeError** exception.
8. Set *opt*.[[nu]] to *numberingSystem*.
9. Let *r* be ResolveLocale(%Intl.NumberFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.NumberFormat%.[[RelevantExtensionKeys]], %Intl.NumberFormat%.[[LocaleData]]).
10. Set *numberFormat*.[[Locale]] to *r*.[[Locale]].
11. Set *numberFormat*.[[LocaleData]] to *r*.[[LocaleData]].
12. Set *numberFormat*.[[NumberingSystem]] to *r*.[[nu]].
13. Perform ? SetNumberFormatUnitOptions(*numberFormat*, *options*).
14. Let *style* be *numberFormat*.[[Style]].
15. If *style* is **"currency"**, then
    a. Let *currency* be *numberFormat*.[[Currency]].

   b. Let *cDigits* be CurrencyDigits(*currency*).
   c. Let *mnfdDefault* be *cDigits*.
   d. Let *mxfdDefault* be *cDigits*.
16. Else,
   a. Let *mnfdDefault* be 0.
   b. If *style* is **"percent"**, then
      i. Let *mxfdDefault* be 0.
   c. Else,
      i. Let *mxfdDefault* be 3.
17. Let *notation* be ? GetOption(*options*, **"notation"**, STRING, « **"standard"**, **"scientific"**, **"engineering"**, **"compact"** », **"standard"**).
18. Set *numberFormat*.[[Notation]] to *notation*.
19. Perform ? SetNumberFormatDigitOptions(*numberFormat*, *options*, *mnfdDefault*, *mxfdDefault*, *notation*).
20. Let *compactDisplay* be ? GetOption(*options*, **"compactDisplay"**, STRING, « **"short"**, **"long"** », **"short"**).
21. Let *defaultUseGrouping* be **"auto"**.
22. If *notation* is **"compact"**, then
   a. Set *numberFormat*.[[CompactDisplay]] to *compactDisplay*.
   b. Set *defaultUseGrouping* to **"min2"**.
23. NOTE: For historical reasons, the strings **"true"** and **"false"** are accepted and replaced with the default value.
24. Let *useGrouping* be ? GetBooleanOrStringNumberFormatOption(*options*, **"useGrouping"**, « **"min2"**, **"auto"**, **"always"**, **"true"**, **"false"** », *defaultUseGrouping*).
25. If *useGrouping* is **"true"** or *useGrouping* is **"false"**, set *useGrouping* to *defaultUseGrouping*.
26. If *useGrouping* is **true**, set *useGrouping* to **"always"**.
27. Set *numberFormat*.[[UseGrouping]] to *useGrouping*.
28. Let *signDisplay* be ? GetOption(*options*, **"signDisplay"**, STRING, « **"auto"**, **"never"**, **"always"**, **"exceptZero"**, **"negative"** », **"auto"**).
29. Set *numberFormat*.[[SignDisplay]] to *signDisplay*.
30. Return UNUSED.

15.1.3 SetNumberFormatDigitOptions (*intlObj*, *options*, *mnfdDefault*, *mxfdDefault*, *notation*)

The abstract operation SetNumberFormatDigitOptions takes arguments *intlObj* (an Object), *options* (an Object), *mnfdDefault* (an integer), *mxfdDefault* (an integer), and *notation* (a String) and returns either a normal completion containing UNUSED or a throw completion. It populates the internal slots of *intlObj* that affect locale-independent number rounding (see 15.5.3). It performs the following steps when called:

1. Let *mnid* be ? GetNumberOption(*options*, **"minimumIntegerDigits,"**, 1, 21, 1).
2. Let *mnfd* be ? Get(*options*, **"minimumFractionDigits"**).
3. Let *mxfd* be ? Get(*options*, **"maximumFractionDigits"**).
4. Let *mnsd* be ? Get(*options*, **"minimumSignificantDigits"**).
5. Let *mxsd* be ? Get(*options*, **"maximumSignificantDigits"**).
6. Set *intlObj*.[[MinimumIntegerDigits]] to *mnid*.
7. Let *roundingIncrement* be ? GetNumberOption(*options*, **"roundingIncrement"**, 1, 5000, 1).
8. If *roundingIncrement* is not in « 1, 2, 5, 10, 20, 25, 50, 100, 200, 250, 500, 1000, 2000, 2500, 5000 », throw a **RangeError** exception.
9. Let *roundingMode* be ? GetOption(*options*, **"roundingMode"**, STRING, « **"ceil"**, **"floor"**, **"expand"**, **"trunc"**, **"halfCeil"**, **"halfFloor"**, **"halfExpand"**, **"halfTrunc"**, **"halfEven"** », **"halfExpand"**).
10. Let *roundingPriority* be ? GetOption(*options*, **"roundingPriority"**, STRING, « **"auto"**, **"morePrecision"**, **"lessPrecision"** », **"auto"**).
11. Let *trailingZeroDisplay* be ? GetOption(*options*, **"trailingZeroDisplay"**, STRING, « **"auto"**, **"stripIfInteger"** », **"auto"**).
12. NOTE: All fields required by SetNumberFormatDigitOptions have now been read from *options*. The remainder of this AO interprets the options and may throw exceptions.
13. If *roundingIncrement* is not 1, set *mxfdDefault* to *mnfdDefault*.
14. Set *intlObj*.[[RoundingIncrement]] to *roundingIncrement*.
15. Set *intlObj*.[[RoundingMode]] to *roundingMode*.
16. Set *intlObj*.[[TrailingZeroDisplay]] to *trailingZeroDisplay*.
17. If *mnsd* is **undefined** and *mxsd* is **undefined**, let *hasSd* be **false**. Otherwise, let *hasSd* be **true**.
18. If *mnfd* is **undefined** and *mxsd* is **undefined**, let *hasFd* be **false**. Otherwise, let *hasFd* be **true**.

19. Let *needSd* be **true**.
20. Let *needFd* be **true**.
21. If *roundingPriority* is **"auto"**, then
    a. Set *needSd* to *hasSd*.
    b. If *needSd* is **true**, or *hasFd* is **false** and *notation* is **"compact"**, then
        i. Set *needFd* to **false**.
22. If *needSd* is **true**, then
    a. If *hasSd* is **true**, then
        i. Set *intlObj*.[[MinimumSignificantDigits]] to ? DefaultNumberOption(*mnsd*, 1, 21, 1).
        ii. Set *intlObj*.[[MaximumSignificantDigits]] to ? DefaultNumberOption(*mxsd*, *intlObj*.[[MinimumSignificantDigits]], 21, 21).
    b. Else,
        i. Set *intlObj*.[[MinimumSignificantDigits]] to 1.
        ii. Set *intlObj*.[[MaximumSignificantDigits]] to 21.
23. If *needFd* is **true**, then
    a. If *hasFd* is **true**, then
        i. Set *mnfd* to ? DefaultNumberOption(*mnfd*, 0, 100, **undefined**).
        ii. Set *mxfd* to ? DefaultNumberOption(*mxfd*, 0, 100, **undefined**).
        iii. If *mnfd* is **undefined**, set *mnfd* to min(*mnfdDefault*, *mxfd*).
        iv. Else if *mxfd* is **undefined**, set *mxfd* to max(*mxfdDefault*, *mnfd*).
        v. Else if *mnfd* is greater than *mxfd*, throw a **RangeError** exception.
        vi. Set *intlObj*.[[MinimumFractionDigits]] to *mnfd*.
        vii. Set *intlObj*.[[MaximumFractionDigits]] to *mxfd*.
    b. Else,
        i. Set *intlObj*.[[MinimumFractionDigits]] to *mnfdDefault*.
        ii. Set *intlObj*.[[MaximumFractionDigits]] to *mxfdDefault*.
24. If *needSd* is **false** and *needFd* is **false**, then
    a. Set *intlObj*.[[MinimumFractionDigits]] to 0.
    b. Set *intlObj*.[[MaximumFractionDigits]] to 0.
    c. Set *intlObj*.[[MinimumSignificantDigits]] to 1.
    d. Set *intlObj*.[[MaximumSignificantDigits]] to 2.
    e. Set *intlObj*.[[RoundingType]] to MORE-PRECISION.
    f. Set *intlObj*.[[ComputedRoundingPriority]] to **"morePrecision"**.
25. Else if *roundingPriority* is **"morePrecision"**, then
    a. Set *intlObj*.[[RoundingType]] to MORE-PRECISION.
    b. Set *intlObj*.[[ComputedRoundingPriority]] to **"morePrecision"**.
26. Else if *roundingPriority* is **"lessPrecision"**, then
    a. Set *intlObj*.[[RoundingType]] to LESS-PRECISION.
    b. Set *intlObj*.[[ComputedRoundingPriority]] to **"lessPrecision"**.
27. Else if *hasSd* is **true**, then
    a. Set *intlObj*.[[RoundingType]] to SIGNIFICANT-DIGITS.
    b. Set *intlObj*.[[ComputedRoundingPriority]] to **"auto"**.
28. Else,
    a. Set *intlObj*.[[RoundingType]] to FRACTION-DIGITS.
    b. Set *intlObj*.[[ComputedRoundingPriority]] to **"auto"**.
29. If *roundingIncrement* is not 1, then
    a. If *intlObj*.[[RoundingType]] is not FRACTION-DIGITS, throw a **TypeError** exception.
    b. If *intlObj*.[[MaximumFractionDigits]] is not equal to *intlObj*.[[MinimumFractionDigits]], throw a **RangeError** exception.
30. Return UNUSED.

15.1.4 SetNumberFormatUnitOptions (*intlObj*, *options*)

The abstract operation SetNumberFormatUnitOptions takes arguments *intlObj* (an Intl.NumberFormat) and *options* (an Object) and returns either a normal completion containing UNUSED or a throw completion. It resolves the user-specified options relating to units onto *intlObj*. It performs the following steps when called:

1. Let *style* be ? GetOption(*options*, **"style"**, STRING, « **"decimal"**, **"percent"**, **"currency"**, **"unit"** », **"decimal"**).
2. Set *intlObj*.[[Style]] to *style*.

3. Let *currency* be ? GetOption(*options*, **"currency"**, STRING, EMPTY, **undefined**).
4. If *currency* is **undefined**, then
   a. If *style* is **"currency"**, throw a **TypeError** exception.
5. Else,
   a. If IsWellFormedCurrencyCode(*currency*) is **false**, throw a **RangeError** exception.
6. Let *currencyDisplay* be ? GetOption(*options*, **"currencyDisplay"**, STRING, « **"code"**, **"symbol"**, **"narrowSymbol"**, **"name"** », **"symbol"**).
7. Let *currencySign* be ? GetOption(*options*, **"currencySign"**, STRING, « **"standard"**, **"accounting"** », **"standard"**).
8. Let *unit* be ? GetOption(*options*, **"unit"**, STRING, EMPTY, **undefined**).
9. If *unit* is **undefined**, then
   a. If *style* is **"unit"**, throw a **TypeError** exception.
10. Else,
    a. If IsWellFormedUnitIdentifier(*unit*) is **false**, throw a **RangeError** exception.
11. Let *unitDisplay* be ? GetOption(*options*, **"unitDisplay"**, STRING, « **"short"**, **"narrow"**, **"long"** », **"short"**).
12. If *style* is **"currency"**, then
    a. Set *intlObj*.[[Currency]] to the ASCII-uppercase of *currency*.
    b. Set *intlObj*.[[CurrencyDisplay]] to *currencyDisplay*.
    c. Set *intlObj*.[[CurrencySign]] to *currencySign*.
13. If *style* is **"unit"**, then
    a. Set *intlObj*.[[Unit]] to *unit*.
    b. Set *intlObj*.[[UnitDisplay]] to *unitDisplay*.
14. Return UNUSED.

15.2 Properties of the Intl.NumberFormat Constructor

The Intl.NumberFormat constructor has the following properties:

15.2.1 Intl.NumberFormat.prototype

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

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

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

15.2.3 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 « **"nu"** ».

NOTE 1 Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> describes three locale extension keys that are relevant to number formatting: **"cu"** for currency, **"cf"** for currency format style, and **"nu"** for numbering system. Intl.NumberFormat, however, requires that the currency of a currency format is specified through the currency property in the options objects, and the currency format style of a currency format is specified through the currencySign 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 four number format styles: **"decimal"**, **"percent"**, **"currency"**, and **"unit"**.
- The two fields **"currency"** and **"unit"** noted above must be Records with at least one field, **"fallback"**. The **"currency"** may have additional fields with keys corresponding to currency codes according to 6.3. Each field of **"currency"** must be a Record with fields corresponding to the possible currencyDisplay values: **"code"**, **"symbol"**, **"narrowSymbol"**, and **"name"**. Each of those fields must contain a Record with fields corresponding to the possible currencySign values: **"standard"** or **"accounting"**. The **"unit"** field (of [[LocaleData]].[[<*locale*>]]) may have additional fields beyond the required field **"fallback"** with keys corresponding to core measurement unit identifiers corresponding to 6.6. Each field of **"unit"** must be a Record with fields corresponding to the possible unitDisplay values: **"narrow"**, **"short"**, and **"long"**.
- All of the leaf fields so far described for the patterns tree (**"decimal"**, **"percent"**, great-grandchildren of **"currency"**, and grandchildren of **"unit"**) must be Records with the keys **"positivePattern"**, **"zeroPattern"**, and **"negativePattern"**.
- The value of the aforementioned fields (the sign-dependent pattern fields) must be string values that must contain the substring **"{number}"**. **"positivePattern"** must contain the substring **"{plusSign}"** but not **"{minusSign}"**; **"negativePattern"** must contain the substring **"{minusSign}"** but not **"{plusSign}"**; and **"zeroPattern"** must not contain either **"{plusSign}"** or **"{minusSign}"**. Additionally, the values within the **"percent"** field must also contain the substring **"{percentSign}"**; the values within the **"currency"** field must also contain one or more of the following substrings: **"{currencyCode}"**, **"{currencyPrefix}"**, or **"{currencySuffix}"**; and the values within the **"unit"** field must also contain one or more of the following substrings: **"{unitPrefix}"** or **"{unitSuffix}"**. The pattern strings, when interpreted as a sequence of UTF-16 encoded code points as described in es2024, 6.1.4, must not contain any code points in the General Category "Number, decimal digit" as specified by the Unicode Standard.
- [[LocaleData]].[[<*locale*>]] must also have a [[notationSubPatterns]] field for all locale values *locale*. The value of this field must be a Record, which must have two fields: [[scientific]] and [[compact]]. The [[scientific]] field must be a string value containing the substrings **"{number}"**, **"{scientificSeparator}"**, and **"{scientificExponent}"**. The [[compact]] field must be a Record with two fields: **"short"** and **"long"**. Each of these fields must be a Record with integer keys corresponding to all discrete magnitudes the implementation supports for compact notation. Each of these fields must be a string value which may contain the substring **"{number}"**. Strings descended from **"short"** must contain the substring **"{compactSymbol}"**, and strings descended from **"long"** must contain the substring **"{compactName}"**.

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

15.3 Properties of the Intl.NumberFormat Prototype Object

The Intl.NumberFormat prototype object is itself an ordinary object. *%Intl.NumberFormat.prototype%* 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.

15.3.1 Intl.NumberFormat.prototype.constructor

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

15.3.2 Intl.NumberFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.NumberFormat"**.

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

15.3.3 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 the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Set *nf* to ? UnwrapNumberFormat(*nf*).
3. Perform ? RequireInternalSlot(*nf*, [[InitializedNumberFormat]]).
4. If *nf*.[[BoundFormat]] is **undefined**, then
   a. Let *F* be a new built-in function object as defined in Number Format Functions (15.5.2).
   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.

15.3.4 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. Perform ? RequireInternalSlot(*nf*, [[InitializedNumberFormat]]).
3. Let *x* be ? ToIntlMathematicalValue(*value*).
4. Return FormatNumericToParts(*nf*, *x*).

15.3.5 Intl.NumberFormat.prototype.formatRange (*start*, *end*)

When the **`formatRange`** method is called with arguments *start* and *end*, the following steps are taken:

1. Let *nf* be the **this** value.
2. Perform ? RequireInternalSlot(*nf*, [[InitializedNumberFormat]]).
3. If *start* is **undefined** or *end* is **undefined**, throw a **TypeError** exception.
4. Let *x* be ? ToIntlMathematicalValue(*start*).
5. Let *y* be ? ToIntlMathematicalValue(*end*).
6. Return ? FormatNumericRange(*nf*, *x*, *y*).

15.3.6 Intl.NumberFormat.prototype.formatRangeToParts (*start*, *end*)

When the **`formatRangeToParts`** method is called with arguments *start* and *end*, the following steps are taken:

1. Let *nf* be the **this** value.
2. Perform ? RequireInternalSlot(*nf*, [[InitializedNumberFormat]]).
3. If *start* is **undefined** or *end* is **undefined**, throw a **TypeError** exception.
4. Let *x* be ? ToIntlMathematicalValue(*start*).
5. Let *y* be ? ToIntlMathematicalValue(*end*).
6. Return ? FormatNumericRangeToParts(*nf*, *x*, *y*).

15.3.7 Intl.NumberFormat.prototype.resolvedOptions ( )

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

1. Let *nf* be the **this** value.
2. If the implementation supports the normative optional constructor mode of 4.3 Note 1, then
   a. Set *nf* to ? UnwrapNumberFormat(*nf*).

3. Perform ? RequireInternalSlot(*nf*, [[InitializedNumberFormat]]).
4. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
5. For each row of Table 12, 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. If there is a Conversion value in the current row, then
         1. Assert: The Conversion value of the current row is NUMBER.
         2. Set *v* to 𝔽(*v*).
      ii. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 12: Resolved Options of NumberFormat Instances

Internal Slot	Property	Conversion
[[Locale]]	**"locale"**	
[[NumberingSystem]]	**"numberingSystem"**	
[[Style]]	**"style"**	
[[Currency]]	**"currency"**	
[[CurrencyDisplay]]	**"currencyDisplay"**	
[[CurrencySign]]	**"currencySign"**	
[[Unit]]	**"unit"**	
[[UnitDisplay]]	**"unitDisplay"**	
[[MinimumIntegerDigits]]	**"minimumIntegerDigits"**	NUMBER
[[MinimumFractionDigits]]	**"minimumFractionDigits"**	NUMBER
[[MaximumFractionDigits]]	**"maximumFractionDigits"**	NUMBER
[[MinimumSignificantDigits]]	**"minimumSignificantDigits"**	NUMBER
[[MaximumSignificantDigits]]	**"maximumSignificantDigits"**	NUMBER
[[UseGrouping]]	**"useGrouping"**	
[[Notation]]	**"notation"**	
[[CompactDisplay]]	**"compactDisplay"**	
[[SignDisplay]]	**"signDisplay"**	
[[RoundingIncrement]]	**"roundingIncrement"**	NUMBER
[[RoundingMode]]	**"roundingMode"**	
[[ComputedRoundingPriority]]	**"roundingPriority"**	
[[TrailingZeroDisplay]]	**"trailingZeroDisplay"**	

15.4 Properties of Intl.NumberFormat Instances

Intl.NumberFormat instances are ordinary objects that inherit properties from %Intl.NumberFormat.prototype%.

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.
- [[LocaleData]] is a Record representing the data available to the implementation for formatting. It is the value of an entry in %Intl.NumberFormat%.[[LocaleData]] associated with either the value of [[Locale]] or a prefix thereof.
- [[NumberingSystem]] is a String value with the "type" given in Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> for the numbering system used for formatting.
- [[Style]] is one of the String values **"decimal"**, **"currency"**, **"percent"**, or **"unit"**, identifying the type of quantity being measured.
- [[Currency]] is a String value with the currency code identifying the currency to be used if formatting with the **"currency"** unit type. It is only used when [[Style]] has the value **"currency"**.
- [[CurrencyDisplay]] is one of the String values **"code"**, **"symbol"**, **"narrowSymbol"**, 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"**.
- [[CurrencySign]] is one of the String values **"standard"** or **"accounting"**, specifying whether to render negative numbers in accounting format, often signified by parenthesis. It is only used when [[Style]] has the value **"currency"** and when [[SignDisplay]] is not **"never"**.
- [[Unit]] is a core unit identifier. It is only used when [[Style]] has the value **"unit"**.
- [[UnitDisplay]] is one of the String values **"short"**, **"narrow"**, or **"long"**, specifying whether to display the unit as a symbol, narrow symbol, or localized long name if formatting with the **"unit"** style. It is only used when [[Style]] has the value **"unit"**.
- [[MinimumIntegerDigits]] is a non-negative integer indicating the minimum integer digits to be used. Numbers will be padded with leading zeroes if necessary.
- [[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integers indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary. These properties are only used when [[RoundingType]] is `FRACTION-DIGITS`, `MORE-PRECISION`, or `LESS-PRECISION`.
- [[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] are positive integers indicating the minimum and maximum fraction digits to be shown. If present, the formatter uses however many fraction digits are required to display the specified number of significant digits. These properties are only used when [[RoundingType]] is `SIGNIFICANT-DIGITS`, `MORE-PRECISION`, or `LESS-PRECISION`.
- [[UseGrouping]] is a Boolean or String value indicating the conditions under which a grouping separator should be used. The positions of grouping separators, and whether to display grouping separators for a formatted number, is implementation-defined. A value **"always"** hints the implementation to display grouping separators if possible; **"min2"**, if there are at least 2 digits in a group; **"auto"**, if the locale prefers to use grouping separators for the formatted number. A value **false** disables grouping separators.
- [[RoundingType]] is one of the values `FRACTION-DIGITS`, `SIGNIFICANT-DIGITS`, `MORE-PRECISION`, or `LESS-PRECISION`, indicating which rounding strategy to use. If `FRACTION-DIGITS`, formatted numbers are rounded according to [[MinimumFractionDigits]] and [[MaximumFractionDigits]], as described above. If `SIGNIFICANT-DIGITS`, formatted numbers are rounded according to [[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] as described above. If `MORE-PRECISION` or `LESS-PRECISION`, all four of those settings are used, with specific rules for disambiguating when to use one set versus the other. [[RoundingType]] is derived from the **"roundingPriority"** option.
- [[ComputedRoundingPriority]] is one of the String values **"auto"**, **"morePrecision"**, or **"lessPrecision"**. It is only used in 15.3.7 to convert [[RoundingType]] back to a valid **"roundingPriority"** option.
- [[Notation]] is one of the String values **"standard"**, **"scientific"**, **"engineering"**, or **"compact"**, specifying whether the formatted number should be displayed without scaling, scaled to the units place with the power of ten in scientific notation, scaled to the nearest thousand with the power of ten in scientific notation, or scaled to the nearest locale-dependent compact decimal notation power of ten with the corresponding compact decimal notation affix.
- [[CompactDisplay]] is one of the String values **"short"** or **"long"**, specifying whether to display compact notation affixes in short form ("5K") or long form ("5 thousand") if formatting with the **"compact"** notation. It is only used when [[Notation]] has the value **"compact"**.
- [[SignDisplay]] is one of the String values **"auto"**, **"always"**, **"never"**, **"exceptZero"**, or **"negative"**, specifying when to include a sign (with non-**"auto"** options respectively corresponding with inclusion always, never, only for non-zero numbers, or only for non-zero negative numbers). In scientific notation, this slot affects the sign display of the mantissa but not the exponent.
- [[RoundingIncrement]] is an integer that evenly divides `10`, `100`, `1000`, or `10000` into tenths, fifths, quarters,

or halves. It indicates the increment at which rounding should take place relative to the calculated rounding magnitude. For example, if [[MaximumFractionDigits]] is 2 and [[RoundingIncrement]] is 5, then formatted numbers are rounded to the nearest `0.05` ("nickel rounding").

- [[RoundingMode]] is one of the String values in the Identifier column of Table 13, specifying which rounding mode to use.
- [[TrailingZeroDisplay]] is one of the String values **"auto"** or **"stripIfInteger"**, indicating whether to strip trailing zeros if the formatted number is an integer (i.e., has no non-zero fraction digit).

Table 13: Rounding modes in Intl.NumberFormat

Identifier	Description	Examples: Round to `0` fraction digits				
		-1.5	`0.4`	`0.5`	`0.6`	1.5
"ceil"	Toward positive infinity	⬆ [-1]	⬆ [1]	⬆ [1]	⬆ [1]	⬆ [2]
"floor"	Toward negative infinity	⬇ [-2]	⬇ [0]	⬇ [0]	⬇ [0]	⬇ [1]
"expand"	Away from zero	⬇ [-2]	⬆ [1]	⬆ [1]	⬆ [1]	⬆ [2]
"trunc"	Toward zero	⬆ [-1]	⬇ [0]	⬇ [0]	⬇ [0]	⬇ [1]
"halfCeil"	Ties toward positive infinity	⬆ [-1]	⬇ [0]	⬆ [1]	⬆ [1]	⬆ [2]
"halfFloor"	Ties toward negative infinity	⬇ [-2]	⬇ [0]	⬇ [0]	⬆ [1]	⬇ [1]
"halfExpand"	Ties away from zero	⬇ [-2]	⬇ [0]	⬆ [1]	⬆ [1]	⬆ [2]
"halfTrunc"	Ties toward zero	⬆ [-1]	⬇ [0]	⬇ [0]	⬆ [1]	⬇ [1]
"halfEven"	Ties toward an even rounding increment multiple	⬇ [-2]	⬇ [0]	⬇ [0]	⬆ [1]	⬆ [2]

NOTE The examples are illustrative of the unique behaviour of each option. ⬆ means "resolves toward positive infinity"; ⬇ means "resolves toward negative infinity".

Finally, Intl.NumberFormat instances have a [[BoundFormat]] internal slot that caches the function returned by the format accessor (15.3.3).

15.5 Abstract Operations for NumberFormat Objects

15.5.1 CurrencyDigits (*currency*)

The abstract operation CurrencyDigits takes argument *currency* (a String) and returns a non-negative integer. It performs the following steps when called:

1. Assert: IsWellFormedCurrencyCode(*currency*) is **true**.
2. Assert: *currency* is equal to the ASCII-uppercase of *currency*.
3. 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.

15.5.2 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 ? ToIntlMathematicalValue(*value*).
5. Return FormatNumeric(*nf*, *x*).

The **"length"** property of a Number format function is **1**$_\mathbb{F}$.

15.5.3 FormatNumericToString (*intlObject*, *x*)

The abstract operation FormatNumericToString takes arguments *intlObject* (an Object) and *x* (a mathematical value or NEGATIVE-ZERO) and returns a Record with fields [[RoundedNumber]] (a mathematical value or NEGATIVE-ZERO) and [[FormattedString]] (a String). It rounds *x* to an Intl mathematical value according to the internal slots of *intlObject*. The [[RoundedNumber]] field contains the rounded result value and the [[FormattedString]] field contains a String value representation of that result formatted according to the internal slots of *intlObject*. It performs the following steps when called:

1. Assert: *intlObject* has [[RoundingMode]], [[RoundingType]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[RoundingIncrement]], and [[TrailingZeroDisplay]] internal slots.
2. If *x* is NEGATIVE-ZERO, then
   a. Let *sign* be NEGATIVE.
   b. Set *x* to 0.
3. Else,
   a. Assert: *x* is a mathematical value.
   b. If *x* < 0, let *sign* be NEGATIVE; else let *sign* be POSITIVE.
   c. If *sign* is NEGATIVE, then
      i. Set *x* to -*x*.
4. Let *unsignedRoundingMode* be GetUnsignedRoundingMode(*intlObject*.[[RoundingMode]], *sign*).
5. If *intlObject*.[[RoundingType]] is SIGNIFICANT-DIGITS, then
   a. Let *result* be ToRawPrecision(*x*, *intlObject*.[[MinimumSignificantDigits]], *intlObject*.[[MaximumSignificantDigits]], *unsignedRoundingMode*).
6. Else if *intlObject*.[[RoundingType]] is FRACTION-DIGITS, then
   a. Let *result* be ToRawFixed(*x*, *intlObject*.[[MinimumFractionDigits]], *intlObject*.[[MaximumFractionDigits]], *intlObject*.[[RoundingIncrement]], *unsignedRoundingMode*).
7. Else,
   a. Let *sResult* be ToRawPrecision(*x*, *intlObject*.[[MinimumSignificantDigits]], *intlObject*.[[MaximumSignificantDigits]], *unsignedRoundingMode*).
   b. Let *fResult* be ToRawFixed(*x*, *intlObject*.[[MinimumFractionDigits]], *intlObject*.[[MaximumFractionDigits]], *intlObject*.[[RoundingIncrement]], *unsignedRoundingMode*).
   c. If *intlObject*.[[RoundingType]] is MORE-PRECISION, then
      i. If *sResult*.[[RoundingMagnitude]] ≤ *fResult*.[[RoundingMagnitude]], then
         1. Let *result* be *sResult*.
      ii. Else,
         1. Let *result* be *fResult*.
   d. Else,
      i. Assert: *intlObject*.[[RoundingType]] is LESS-PRECISION.
      ii. If *sResult*.[[RoundingMagnitude]] ≤ *fResult*.[[RoundingMagnitude]], then
         1. Let *result* be *fResult*.
      iii. Else,
         1. Let *result* be *sResult*.
8. Set *x* to *result*.[[RoundedNumber]].
9. Let *string* be *result*.[[FormattedString]].
10. If *intlObject*.[[TrailingZeroDisplay]] is **"stripIfInteger"** and *x* modulo 1 = 0, then
    a. Let *i* be StringIndexOf(*string*, **"."**, 0).
    b. If *i* ≠ -1, set *string* to the substring of *string* from 0 to *i*.
11. Let *int* be *result*.[[IntegerDigitsCount]].
12. Let *minInteger* be *intlObject*.[[MinimumIntegerDigits]].
13. If *int* < *minInteger*, then
    a. Let *forwardZeros* be the String consisting of *minInteger* - *int* occurrences of the code unit 0x0030 (DIGIT ZERO).
    b. Set *string* to the string-concatenation of *forwardZeros* and *string*.

14. If *sign* is NEGATIVE, then
    a. If *x* is 0, set *x* to NEGATIVE-ZERO. Otherwise, set *x* to -*x*.
15. Return the Record { [[RoundedNumber]]: *x*, [[FormattedString]]: *string* }.

15.5.4 PartitionNumberPattern (*numberFormat*, *x*)

The abstract operation PartitionNumberPattern takes arguments *numberFormat* (an object initialized as a NumberFormat) and *x* (an Intl mathematical value) and returns a List of Records with fields [[Type]] (a String) and [[Value]] (a String). It creates the parts representing the mathematical value of *x* according to the effective locale and the formatting options of *numberFormat*. It performs the following steps when called:

1. Let *exponent* be 0.
2. If *x* is NOT-A-NUMBER, then
   a. Let *n* be an implementation- and locale-dependent (ILD) String value indicating the **NaN** value.
3. Else if *x* is POSITIVE-INFINITY, then
   a. Let *n* be an ILD String value indicating positive infinity.
4. Else if *x* is NEGATIVE-INFINITY, then
   a. Let *n* be an ILD String value indicating negative infinity.
5. Else,
   a. If *x* is not NEGATIVE-ZERO, then
      i. Assert: *x* is a mathematical value.
      ii. If *numberFormat*.[[Style]] is **"percent"**, set *x* be 100 × *x*.
      iii. Set *exponent* to ComputeExponent(*numberFormat*, *x*).
      iv. Set *x* to *x* × $10^{-exponent}$.
   b. Let *formatNumberResult* be FormatNumericToString(*numberFormat*, *x*).
   c. Let *n* be *formatNumberResult*.[[FormattedString]].
   d. Set *x* to *formatNumberResult*.[[RoundedNumber]].
6. Let *pattern* be GetNumberFormatPattern(*numberFormat*, *x*).
7. Let *result* be a new empty List.
8. Let *patternParts* be PartitionPattern(*pattern*).
9. For each Record { [[Type]], [[Value]] } *patternPart* of *patternParts*, do
   a. Let *p* be *patternPart*.[[Type]].
   b. If *p* is **"literal"**, then
      i. Append the Record { [[Type]]: **"literal"**, [[Value]]: *patternPart*.[[Value]] } to *result*.
   c. Else if *p* is equal to **"number"**, then
      i. Let *notationSubParts* be PartitionNotationSubPattern(*numberFormat*, *x*, *n*, *exponent*).
      ii. For each Record { [[Type]], [[Value]] } *subPart* of *notationSubParts*, do
         1. Append *subPart* to *result*.
   d. Else if *p* is equal to **"plusSign"**, then
      i. Let *plusSignSymbol* be the ILND String representing the plus sign.
      ii. Append the Record { [[Type]]: **"plusSign"**, [[Value]]: *plusSignSymbol* } to *result*.
   e. Else if *p* is equal to **"minusSign"**, then
      i. Let *minusSignSymbol* be the ILND String representing the minus sign.
      ii. Append the Record { [[Type]]: **"minusSign"**, [[Value]]: *minusSignSymbol* } to *result*.
   f. 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 the Record { [[Type]]: **"percentSign"**, [[Value]]: *percentSignSymbol* } to *result*.
   g. Else if *p* is equal to **"unitPrefix"** and *numberFormat*.[[Style]] is **"unit"**, then
      i. Let *unit* be *numberFormat*.[[Unit]].
      ii. Let *unitDisplay* be *numberFormat*.[[UnitDisplay]].
      iii. Let *mu* be an ILD String value representing *unit* before *x* in *unitDisplay* form, which may depend on *x* in languages having different plural forms.
      iv. Append the Record { [[Type]]: **"unit"**, [[Value]]: *mu* } to *result*.
   h. Else if *p* is equal to **"unitSuffix"** and *numberFormat*.[[Style]] is **"unit"**, then
      i. Let *unit* be *numberFormat*.[[Unit]].
      ii. Let *unitDisplay* be *numberFormat*.[[UnitDisplay]].
      iii. Let *mu* be an ILD String value representing *unit* after *x* in *unitDisplay* form, which may depend on *x* in languages having different plural forms.
      iv. Append the Record { [[Type]]: **"unit"**, [[Value]]: *mu* } to *result*.
   i. Else if *p* is equal to **"currencyCode"** and *numberFormat*.[[Style]] is **"currency"**, then

i. Let *currency* be *numberFormat*.[[Currency]].
ii. Let *cd* be *currency*.
iii. Append the Record { [[Type]]: **"currency"**, [[Value]]: *cd* } to *result*.

j. Else if *p* is equal to **"currencyPrefix"** and *numberFormat*.[[Style]] is **"currency"**, then
i. Let *currency* be *numberFormat*.[[Currency]].
ii. Let *currencyDisplay* be *numberFormat*.[[CurrencyDisplay]].
iii. Let *cd* be an ILD String value representing *currency* before *x* in *currencyDisplay* form, which may depend on *x* in languages having different plural forms.
iv. Append the Record { [[Type]]: **"currency"**, [[Value]]: *cd* } to *result*.

k. Else if *p* is equal to **"currencySuffix"** and *numberFormat*.[[Style]] is **"currency"**, then
i. Let *currency* be *numberFormat*.[[Currency]].
ii. Let *currencyDisplay* be *numberFormat*.[[CurrencyDisplay]].
iii. Let *cd* be an ILD String value representing *currency* after *x* in *currencyDisplay* form, which may depend on *x* in languages having different plural forms. If the implementation does not have such a representation of *currency*, use *currency* itself.
iv. Append the Record { [[Type]]: **"currency"**, [[Value]]: *cd* } to *result*.

l. Else,
i. Let *unknown* be an ILND String based on *x* and *p*.
ii. Append the Record { [[Type]]: **"unknown"**, [[Value]]: *unknown* } to *result*.

10. Return *result*.

15.5.5 PartitionNotationSubPattern (*numberFormat*, *x*, *n*, *exponent*)

The abstract operation PartitionNotationSubPattern takes arguments *numberFormat* (an Intl.NumberFormat), *x* (an Intl mathematical value), *n* (a String), and *exponent* (an integer) and returns a List of Records with fields [[Type]] (a String) and [[Value]] (a String). *x* is an Intl mathematical value after rounding is applied and *n* is an intermediate formatted string. It creates the corresponding parts for the number and notation according to the effective locale and the formatting options of *numberFormat*. It performs the following steps when called:

1. Let *result* be a new empty List.
2. If *x* is NOT-A-NUMBER, then
   a. Append the Record { [[Type]]: **"nan"**, [[Value]]: *n* } to *result*.
3. Else if *x* is POSITIVE-INFINITY or NEGATIVE-INFINITY, then
   a. Append the Record { [[Type]]: **"infinity"**, [[Value]]: *n* } to *result*.
4. Else,
   a. Let *notationSubPattern* be GetNotationSubPattern(*numberFormat*, *exponent*).
   b. Let *patternParts* be PartitionPattern(*notationSubPattern*).
   c. For each Record { [[Type]], [[Value]] } *patternPart* of *patternParts*, do
      i. Let *p* be *patternPart*.[[Type]].
      ii. If *p* is **"literal"**, then
         1. Append the Record { [[Type]]: **"literal"**, [[Value]]: *patternPart*.[[Value]] } to *result*.
      iii. Else if *p* is equal to **"number"**, then
         1. If the *numberFormat*.[[NumberingSystem]] matches one of the values in the Numbering System column of Table 14 below, then
            a. Let *digits* be a List whose elements are the code points specified in the Digits column of the matching row in Table 14.
            b. Assert: The length of *digits* is 10.
            c. Let *transliterated* be the empty String.
            d. Let *len* be the length of *n*.
            e. Let *position* be 0.
            f. Repeat, while *position* < *len*,
               i. Let *c* be the code unit at index *position* within *n*.
               ii. If 0x0030 ≤ *c* ≤ 0x0039, then
                  i. NOTE: *c* is an ASCII digit.
                  ii. Let *i* be *c* - 0x0030.
                  iii. Set *c* to CodePointsToString(« *digits*[*i*] »).
               iii. Set *transliterated* to the string-concatenation of *transliterated* and *c*.
               iv. Set *position* to *position* + 1.
            g. Set *n* to *transliterated*.
         2. Else,

a. Use an implementation dependent algorithm to map *n* to the appropriate representation of *n* in the given numbering system.
3. Let *decimalSepIndex* be StringIndexOf(*n*, **"."**, 0).
4. 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*.
5. Else,
   a. Let *integer* be *n*.
   b. Let *fraction* be **undefined**.
6. If the *numberFormat*.[[UseGrouping]] is **false**, then
   a. Append the Record { [[Type]]: **"integer"**, [[Value]]: *integer* } to *result*.
7. Else,
   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*, which may depend on the value of *numberFormat*.[[UseGrouping]].
   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 the Record { [[Type]]: **"integer"**, [[Value]]: *integerGroup* } to *result*.
      iii. If *groups* List is not empty, then
         i. Append the Record { [[Type]]: **"group"**, [[Value]]: *groupSepSymbol* } to *result*.
8. If *fraction* is not **undefined**, then
   a. Let *decimalSepSymbol* be the ILND String representing the decimal separator.
   b. Append the Record { [[Type]]: **"decimal"**, [[Value]]: *decimalSepSymbol* } to *result*.
   c. Append the Record { [[Type]]: **"fraction"**, [[Value]]: *fraction* } to *result*.

iv. Else if *p* is equal to **"compactSymbol"**, then
   1. Let *compactSymbol* be an ILD string representing *exponent* in short form, which may depend on *x* in languages having different plural forms. The implementation must be able to provide this string, or else the pattern would not have a **"{compactSymbol}"** placeholder.
   2. Append the Record { [[Type]]: **"compact"**, [[Value]]: *compactSymbol* } to *result*.

v. Else if *p* is equal to **"compactName"**, then
   1. Let *compactName* be an ILD string representing *exponent* in long form, which may depend on *x* in languages having different plural forms. The implementation must be able to provide this string, or else the pattern would not have a **"{compactName}"** placeholder.
   2. Append the Record { [[Type]]: **"compact"**, [[Value]]: *compactName* } to *result*.

vi. Else if *p* is equal to **"scientificSeparator"**, then
   1. Let *scientificSeparator* be the ILND String representing the exponent separator.
   2. Append the Record { [[Type]]: **"exponentSeparator"**, [[Value]]: *scientificSeparator* } to *result*.

vii. Else if *p* is equal to **"scientificExponent"**, then
   1. If *exponent* < 0, then
      a. Let *minusSignSymbol* be the ILND String representing the minus sign.
      b. Append the Record { [[Type]]: **"exponentMinusSign"**, [[Value]]: *minusSignSymbol* } to *result*.
      c. Let *exponent* be -*exponent*.
   2. Let *exponentResult* be ToRawFixed(*exponent*, 0, 0, 1, **undefined**).
   3. Append the Record { [[Type]]: **"exponentInteger"**, [[Value]]: *exponentResult*.[[FormattedString]] } to *result*.

viii. Else,
   1. Let *unknown* be an ILND String based on *x* and *p*.
   2. Append the Record { [[Type]]: **"unknown"**, [[Value]]: *unknown* } to *result*.

5. Return *result*.

Table 14: Numbering systems with simple digit mappings

Numbering System	Digits
adlm	U+1E950 to U+1E959
ahom	U+11730 to U+11739
arab	U+0660 to U+0669
arabext	U+06F0 to U+06F9
bali	U+1B50 to U+1B59
beng	U+09E6 to U+09EF
bhks	U+11C50 to U+11C59
brah	U+11066 to U+1106F
cakm	U+11136 to U+1113F
cham	U+AA50 to U+AA59
deva	U+0966 to U+096F
diak	U+11950 to U+11959
fullwide	U+FF10 to U+FF19
gong	U+11DA0 to U+11DA9
gonm	U+11D50 to U+11D59
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
hmng	U+16B50 to U+16B59
hmnp	U+1E140 to U+1E149
java	U+A9D0 to U+A9D9
kali	U+A900 to U+A909
kawi	U+11F50 to U+11F59
khmr	U+17E0 to U+17E9
knda	U+0CE6 to U+0CEF
lana	U+1A80 to U+1A89
lanatham	U+1A90 to U+1A99
laoo	U+0ED0 to U+0ED9
latn	U+0030 to U+0039
lepc	U+1C40 to U+1C49
limb	U+1946 to U+194F
mathbold	U+1D7CE to U+1D7D7

Table 14: Numbering systems with simple digit mappings *(continued)*

Numbering System	Digits
mathdbl	U+1D7D8 to U+1D7E1
mathmono	U+1D7F6 to U+1D7FF
mathsanb	U+1D7EC to U+1D7F5
mathsans	U+1D7E2 to U+1D7EB
mlym	U+0D66 to U+0D6F
modi	U+11650 to U+11659
mong	U+1810 to U+1819
mroo	U+16A60 to U+16A69
mtei	U+ABF0 to U+ABF9
mymr	U+1040 to U+1049
mymrshan	U+1090 to U+1099
mymrtlng	U+A9F0 to U+A9F9
nagm	U+1E4F0 to U+1E4F9
newa	U+11450 to U+11459
nkoo	U+07C0 to U+07C9
olck	U+1C50 to U+1C59
orya	U+0B66 to U+0B6F
osma	U+104A0 to U+104A9
rohg	U+10D30 to U+10D39
saur	U+A8D0 to U+A8D9
segment	U+1FBF0 to U+1FBF9
shrd	U+111D0 to U+111D9
sind	U+112F0 to U+112F9
sinh	U+0DE6 to U+0DEF
sora	U+110F0 to U+110F9
sund	U+1BB0 to U+1BB9
takr	U+116C0 to U+116C9
talu	U+19D0 to U+19D9
tamldec	U+0BE6 to U+0BEF
telu	U+0C66 to U+0C6F
thai	U+0E50 to U+0E59
tibt	U+0F20 to U+0F29

Table 14: Numbering systems with simple digit mappings *(continued)*

Numbering System	Digits
tirh	U+114D0 to U+114D9
tnsa	U+16AC0 to U+16AC9
vaii	U+A620 to U+A629
wara	U+118E0 to U+118E9
wcho	U+1E2F0 to U+1E2F9

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 code points 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 https://cldr.unicode.org/).

15.5.6 FormatNumeric (*numberFormat*, *x*)

The abstract operation FormatNumeric takes arguments *numberFormat* (an Intl.NumberFormat) and *x* (an Intl mathematical value) and returns a String. It performs the following steps when called:

1. Let *parts* be PartitionNumberPattern(*numberFormat*, *x*).
2. Let *result* be the empty String.
3. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

15.5.7 FormatNumericToParts (*numberFormat*, *x*)

The abstract operation FormatNumericToParts takes arguments *numberFormat* (an Intl.NumberFormat) and *x* (an Intl mathematical value) and returns an Array. It performs the following steps when called:

1. Let *parts* be PartitionNumberPattern(*numberFormat*, *x*).
2. Let *result* be ! ArrayCreate(0).
3. Let *n* be 0.
4. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
   a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
   d. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString($\mathbb{F}$(*n*)), *O*).
   e. Increment *n* by 1.
5. Return *result*.

15.5.8 ToRawPrecision (*x*, *minPrecision*, *maxPrecision*, *unsignedRoundingMode*)

The abstract operation ToRawPrecision takes arguments *x* (non-negative mathematical value), *minPrecision* (an integer in the inclusive interval from 1 to 21), *maxPrecision* (an integer in the inclusive interval from 1 to 21), and *unsignedRoundingMode* (a specification type from the Unsigned Rounding Mode column of Table 15, or **undefined**) and returns a Record with fields [[FormattedString]] (a String), [[RoundedNumber]] (a mathematical value), [[IntegerDigitsCount]] (an integer), and [[RoundingMagnitude]] (an integer).

It involves solving the following equation, which returns a valid mathematical value given integer inputs:

$$\text{ToRawPrecisionFn}(n, e, p) = n \times 10^{e-p+1}$$
$$\text{where } 10^{p-1} \le n < 10^{p}$$

It performs the following steps when called:

1. Let *p* be *maxPrecision*.
2. If *x* = 0, then
   a. Let *m* be the String consisting of *p* occurrences of the code unit 0x0030 (DIGIT ZERO).
   b. Let *e* be 0.
   c. Let *xFinal* be 0.
3. Else,
   a. Let *n1* and *e1* each be an integer and *r1* a mathematical value, with *r1* = ToRawPrecisionFn(*n1*, *e1*, *p*), such that *r1* ≤ *x* and *r1* is maximized.
   b. Let *n2* and *e2* each be an integer and *r2* a mathematical value, with *r2* = ToRawPrecisionFn(*n2*, *e2*, *p*), such that *r2* ≥ *x* and *r2* is minimized.
   c. Let *r* be ApplyUnsignedRoundingMode(*x*, *r1*, *r2*, *unsignedRoundingMode*).
   d. If *r* is *r1*, then
      i. Let *n* be *n1*.
      ii. Let *e* be *e1*.
      iii. Let *xFinal* be *r1*.
   e. Else,
      i. Let *n* be *n2*.
      ii. Let *e* be *e2*.
      iii. Let *xFinal* be *r2*.
   f. 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* - 1), then
   a. Set *m* to the string-concatenation of *m* and *e* - *p* + 1 occurrences of the code unit 0x0030 (DIGIT ZERO).
   b. Let *int* be *e* + 1.
5. Else if *e* ≥ 0, then
   a. Set *m* to the string-concatenation of the first *e* + 1 code units of *m*, the code unit 0x002E (FULL STOP), and the remaining *p* - (*e* + 1) code units of *m*.
   b. Let *int* be *e* + 1.
6. Else,
   a. Assert: *e* < 0.
   b. Set *m* to the string-concatenation of **"0."**, -(*e* + 1) occurrences of the code unit 0x0030 (DIGIT ZERO), and *m*.
   c. Let *int* be 1.
7. If *m* contains the code unit 0x002E (FULL STOP) and *maxPrecision* > *minPrecision*, then
   a. Let *cut* be *maxPrecision* - *minPrecision*.
   b. Repeat, while *cut* > 0 and the last code unit of *m* is 0x0030 (DIGIT ZERO),
      i. Remove the last code unit from *m*.
      ii. Set *cut* to *cut* - 1.
   c. If the last code unit of *m* is 0x002E (FULL STOP), then
      i. Remove the last code unit from *m*.
8. Return the Record { [[FormattedString]]: *m*, [[RoundedNumber]]: *xFinal*, [[IntegerDigitsCount]]: *int*, [[RoundingMagnitude]]: *e*–*p*+1 }.

15.5.9 ToRawFixed (*x*, *minFraction*, *maxFraction*, *roundingIncrement*, *unsignedRoundingMode*)

The abstract operation ToRawFixed takes arguments *x* (non-negative mathematical value), *minFraction* (an integer in the inclusive interval from 0 to 100), *maxFraction* (an integer in the inclusive interval from 0 to 100), *roundingIncrement* (an integer), and *unsignedRoundingMode* (a specification type from the Unsigned Rounding Mode column of Table 15, or **undefined**) and returns a Record with fields [[FormattedString]] (a String), [[RoundedNumber]] (a mathematical value), [[IntegerDigitsCount]] (an integer), and [[RoundingMagnitude]] (an integer).

It involves solving the following equation, which returns a valid mathematical value given integer inputs:

$$\text{ToRawFixedFn}(n, f) = n \times 10^{-f}$$

It performs the following steps when called:

1. Let *f* be *maxFraction*.
2. Let *n1* be an integer and *r1* a mathematical value, with *r1* = ToRawFixedFn(*n1*, *f*), such that *n1* modulo *roundingIncrement* = `0`, *r1* ≤ *x*, and *r1* is maximized.
3. Let *n2* be an integer and *r2* a mathematical value, with *r2* = ToRawFixedFn(*n2*, *f*), such that *n2* modulo *roundingIncrement* = `0`, *r2* ≥ *x*, and *r2* is minimized.
4. Let *r* be ApplyUnsignedRoundingMode(*x*, *r1*, *r2*, *unsignedRoundingMode*).
5. If *r* is *r1*, then
   a. Let *n* be *n1*.
   b. Let *xFinal* be *r1*.
6. Else,
   a. Let *n* be *n2*.
   b. Let *xFinal* be *r2*.
7. If *n* = `0`, let *m* be **"0"**. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
8. If *f* ≠ `0`, then
   a. Let *k* be the length of *m*.
   b. If *k* ≤ *f*, then
      i. Let *z* be the String value consisting of *f* + 1 - *k* occurrences of the code unit `0x0030` (DIGIT ZERO).
      ii. Set *m* to the string-concatenation of *z* and *m*.
      iii. Set *k* to *f* + 1.
   c. Let *a* be the first *k* - *f* code units of *m*, and let *b* be the remaining *f* code units of *m*.
   d. Set *m* to the string-concatenation of *a*, **"."**, and *b*.
   e. Let *int* be the length of *a*.
9. Else,
   a. Let *int* be the length of *m*.
10. Let *cut* be *maxFraction* - *minFraction*.
11. Repeat, while *cut* > `0` and the last code unit of *m* is `0x0030` (DIGIT ZERO),
    a. Remove the last code unit from *m*.
    b. Set *cut* to *cut* - 1.
12. If the last code unit of *m* is `0x002E` (FULL STOP), then
    a. Remove the last code unit from *m*.
13. Return the Record { [[FormattedString]]: *m*, [[RoundedNumber]]: *xFinal*, [[IntegerDigitsCount]]: *int*, [[RoundingMagnitude]]: –*f* }.

NORMATIVE OPTIONAL

15.5.10 UnwrapNumberFormat (*nf*)

The abstract operation UnwrapNumberFormat takes argument *nf* (an ECMAScript language value) and returns either a normal completion containing an ECMAScript language value or a throw completion. It returns the NumberFormat instance of its input object, which is either the value itself or a value associated with it by %Intl.NumberFormat% according to the normative optional constructor mode of 4.3 Note 1. It performs the following steps when called:

1. If Type(*nf*) is not Object, throw a **TypeError** exception.
2. If *nf* does not have an [[InitializedNumberFormat]] internal slot and ? OrdinaryHasInstance(%Intl.NumberFormat%, *nf*) is **true**, then
   a. Return ? Get(*nf*, %Intl%.[[FallbackSymbol]]).
3. Return *nf*.

15.5.11 GetNumberFormatPattern (*numberFormat*, *x*)

The abstract operation GetNumberFormatPattern takes arguments *numberFormat* (an Intl.NumberFormat) and *x* (an Intl mathematical value) and returns a String. It considers the resolved unit-related options in the number format object along with the final scaled and rounded number being formatted (an Intl mathematical value) and returns a pattern, a String value as described in 15.2.3. It performs the following steps when called:

1. Let *resolvedLocaleData* be *numberFormat*.[[LocaleData]].
2. Let *patterns* be *resolvedLocaleData*.[[patterns]].
3. Assert: *patterns* is a Record (see 15.2.3).
4. Let *style* be *numberFormat*.[[Style]].
5. If *style* is **"percent"**, then
   a. Set *patterns* to *patterns*.[[percent]].
6. Else if *style* is **"unit"**, then
   a. Let *unit* be *numberFormat*.[[Unit]].
   b. Let *unitDisplay* be *numberFormat*.[[UnitDisplay]].
   c. Set *patterns* to *patterns*.[[unit]].
   d. If *patterns* doesn't have a field [[<*unit*>]], then
      i. Set *unit* to **"fallback"**.
   e. Set *patterns* to *patterns*.[[<*unit*>]].
   f. Set *patterns* to *patterns*.[[<*unitDisplay*>]].
7. Else if *style* is **"currency"**, then
   a. Let *currency* be *numberFormat*.[[Currency]].
   b. Let *currencyDisplay* be *numberFormat*.[[CurrencyDisplay]].
   c. Let *currencySign* be *numberFormat*.[[CurrencySign]].
   d. Set *patterns* to *patterns*.[[currency]].
   e. If *patterns* doesn't have a field [[<*currency*>]], then
      i. Set *currency* to **"fallback"**.
   f. Set *patterns* to *patterns*.[[<*currency*>]].
   g. Set *patterns* to *patterns*.[[<*currencyDisplay*>]].
   h. Set *patterns* to *patterns*.[[<*currencySign*>]].
8. Else,
   a. Assert: *style* is **"decimal"**.
   b. Set *patterns* to *patterns*.[[decimal]].
9. If *x* is NEGATIVE-INFINITY, then
   a. Let *category* be NEGATIVE-NON-ZERO.
10. Else if *x* is NEGATIVE-ZERO, then
    a. Let *category* be NEGATIVE-ZERO.
11. Else if *x* is NOT-A-NUMBER, then
    a. Let *category* be POSITIVE-ZERO.
12. Else if *x* is POSITIVE-INFINITY, then
    a. Let *category* be POSITIVE-NON-ZERO.
13. Else,
    a. Assert: *x* is a mathematical value.
    b. If *x* < 0, then
       i. Let *category* be NEGATIVE-NON-ZERO.
    c. Else if *x* > 0, then
       i. Let *category* be POSITIVE-NON-ZERO.
    d. Else,
       i. Let *category* be POSITIVE-ZERO.
14. Let *signDisplay* be *numberFormat*.[[SignDisplay]].
15. If *signDisplay* is **"never"**, then
    a. Let *pattern* be *patterns*.[[zeroPattern]].
16. Else if *signDisplay* is **"auto"**, then
    a. If *category* is POSITIVE-NON-ZERO or POSITIVE-ZERO, then
       i. Let *pattern* be *patterns*.[[zeroPattern]].
    b. Else,
       i. Let *pattern* be *patterns*.[[negativePattern]].
17. Else if *signDisplay* is **"always"**, then
    a. If *category* is POSITIVE-NON-ZERO or POSITIVE-ZERO, then

        i. Let *pattern* be *patterns*.[[positivePattern]].
    b. Else,
        i. Let *pattern* be *patterns*.[[negativePattern]].
18. Else if *signDisplay* is **"exceptZero"**, then
    a. If *category* is POSITIVE-ZERO or NEGATIVE-ZERO, then
        i. Let *pattern* be *patterns*.[[zeroPattern]].
    b. Else if *category* is POSITIVE-NON-ZERO, then
        i. Let *pattern* be *patterns*.[[positivePattern]].
    c. Else,
        i. Let *pattern* be *patterns*.[[negativePattern]].
19. Else,
    a. Assert: *signDisplay* is **"negative"**.
    b. If *category* is NEGATIVE-NON-ZERO, then
        i. Let *pattern* be *patterns*.[[negativePattern]].
    c. Else,
        i. Let *pattern* be *patterns*.[[zeroPattern]].
20. Return *pattern*.

15.5.12 GetNotationSubPattern (*numberFormat*, *exponent*)

The abstract operation GetNotationSubPattern takes arguments *numberFormat* (an Intl.NumberFormat) and *exponent* (an integer) and returns a String. It considers the resolved notation and *exponent*, and returns a String value for the notation sub pattern as described in 15.2.3. It performs the following steps when called:

1. Let *resolvedLocaleData* be *numberFormat*.[[LocaleData]].
2. Let *notationSubPatterns* be *resolvedLocaleData*.[[notationSubPatterns]].
3. Assert: *notationSubPatterns* is a Record (see 15.2.3).
4. Let *notation* be *numberFormat*.[[Notation]].
5. If *notation* is **"scientific"** or *notation* is **"engineering"**, then
    a. Return *notationSubPatterns*.[[scientific]].
6. Else if *exponent* is not 0, then
    a. Assert: *notation* is **"compact"**.
    b. Let *compactDisplay* be *numberFormat*.[[CompactDisplay]].
    c. Let *compactPatterns* be *notationSubPatterns*.[[compact]].[[<*compactDisplay*>]].
    d. Return *compactPatterns*.[[<*exponent*>]].
7. Else,
    a. Return **"{number}"**.

15.5.13 ComputeExponent (*numberFormat*, *x*)

The abstract operation ComputeExponent takes arguments *numberFormat* (an Intl.NumberFormat) and *x* (a mathematical value) and returns an integer. It computes an exponent (power of ten) by which to scale *x* according to the number formatting settings. It handles cases such as 999 rounding up to 1000, requiring a different exponent. It performs the following steps when called:

1. If *x* = 0, then
    a. Return 0.
2. If *x* < 0, then
    a. Let *x* = -*x*.
3. Let *magnitude* be the base 10 logarithm of *x* rounded down to the nearest integer.
4. Let *exponent* be ComputeExponentForMagnitude(*numberFormat*, *magnitude*).
5. Let *x* be *x* × $10^{-exponent}$.
6. Let *formatNumberResult* be FormatNumericToString(*numberFormat*, *x*).
7. If *formatNumberResult*.[[RoundedNumber]] = 0, then
    a. Return *exponent*.
8. Let *newMagnitude* be the base 10 logarithm of *formatNumberResult*.[[RoundedNumber]] rounded down to the nearest integer.
9. If *newMagnitude* is *magnitude* - *exponent*, then

   a. Return *exponent*.
10. Return ComputeExponentForMagnitude(*numberFormat*, *magnitude* + 1).

15.5.14 ComputeExponentForMagnitude (*numberFormat*, *magnitude*)

The abstract operation ComputeExponentForMagnitude takes arguments *numberFormat* (an Intl.NumberFormat) and *magnitude* (an integer) and returns an integer. It computes an exponent by which to scale a number of the given magnitude (power of ten of the most significant digit) according to the locale and the desired notation (scientific, engineering, or compact). It performs the following steps when called:

1. Let *notation* be *numberFormat*.[[Notation]].
2. If *notation* is **"standard"**, then
   a. Return `0`.
3. Else if *notation* is **"scientific"**, then
   a. Return *magnitude*.
4. Else if *notation* is **"engineering"**, then
   a. Let *thousands* be the greatest integer that is not greater than *magnitude* / 3.
   b. Return *thousands* × 3.
5. Else,
   a. Assert: *notation* is **"compact"**.
   b. Let *exponent* be an implementation- and locale-dependent (ILD) integer by which to scale a number of the given magnitude in compact notation for the current locale.
   c. Return *exponent*.

15.5.15 Runtime Semantics: StringIntlMV

The syntax-directed operation StringIntlMV takes no arguments.

NOTE The conversion of a *StringNumericLiteral* to a Number value is similar overall to the determination of the NumericValue of a *NumericLiteral* (see 12.9.3), but some of the details are different.

It is defined piecewise over the following productions:

StringNumericLiteral ::: *StrWhiteSpace*opt

1. Return `0`.

StringNumericLiteral ::: *StrWhiteSpace*opt *StrNumericLiteral* *StrWhiteSpace*opt

1. Return StringIntlMV of *StrNumericLiteral*.

StrNumericLiteral ::: *NonDecimalIntegerLiteral*

1. Return MV of *NonDecimalIntegerLiteral*.

StrDecimalLiteral ::: **-** *StrUnsignedDecimalLiteral*

1. Let *a* be StringIntlMV of *StrUnsignedDecimalLiteral*.
2. If *a* is `0`, return NEGATIVE-ZERO.
3. If *a* is POSITIVE-INFINITY, return NEGATIVE-INFINITY.
4. Return -*a*.

StrUnsignedDecimalLiteral ::: **`Infinity`**

1. Return POSITIVE-INFINITY.

StrUnsignedDecimalLiteral ::: *DecimalDigits* **.** *DecimalDigits*$_{opt}$ *ExponentPart*$_{opt}$

1. Let *a* be MV of the first *DecimalDigits*.
2. If the second *DecimalDigits* is present, then
   a. Let *b* be MV of the second *DecimalDigits*.
   b. Let *n* be the number of code points in the second *DecimalDigits*.
3. Else,
   a. Let *b* be 0.
   b. Let *n* be 0.
4. If *ExponentPart* is present, let *e* be MV of *ExponentPart*. Otherwise, let *e* be 0.
5. Return $(a + (b \times 10^{-n})) \times 10^{e}$.

StrUnsignedDecimalLiteral ::: **.** *DecimalDigits* *ExponentPart*$_{opt}$

1. Let *b* be MV of *DecimalDigits*.
2. If *ExponentPart* is present, let *e* be MV of *ExponentPart*. Otherwise, let *e* be 0.
3. Let *n* be the number of code points in *DecimalDigits*.
4. Return $b \times 10^{e - n}$.

StrUnsignedDecimalLiteral ::: *DecimalDigits* *ExponentPart*$_{opt}$

1. Let *a* be MV of *DecimalDigits*.
2. If *ExponentPart* is present, let *e* be MV of *ExponentPart*. Otherwise, let *e* be 0.
3. Return $a \times 10^{e}$.

15.5.16 ToIntlMathematicalValue (*value*)

The abstract operation ToIntlMathematicalValue takes argument *value* (an ECMAScript language value) and returns either a normal completion containing an Intl mathematical value or a throw completion. It returns *value* converted to an *Intl mathematical value*, which is a mathematical value together with POSITIVE-INFINITY, NEGATIVE-INFINITY, NOT-A-NUMBER, and NEGATIVE-ZERO. This abstract operation is similar to 7.1.3, but a mathematical value can be returned instead of a Number or BigInt, so that exact decimal values can be represented. It performs the following steps when called:

1. Let *primValue* be ? ToPrimitive(*value*, NUMBER).
2. If Type(*primValue*) is BigInt, return ℝ(*primValue*).
3. If Type(*primValue*) is String, then
   a. Let *str* be *primValue*.
4. Else,
   a. Let *x* be ? ToNumber(*primValue*).
   b. If *x* is **-0**𝔽, return NEGATIVE-ZERO.
   c. Let *str* be Number::toString(*x*, 10).
5. Let *text* be StringToCodePoints(*str*).
6. Let *literal* be ParseText(*text*, *StringNumericLiteral*).
7. If *literal* is a List of errors, return NOT-A-NUMBER.
8. Let *intlMV* be the StringIntlMV of *literal*.
9. If *intlMV* is a mathematical value, then
   a. Let *rounded* be RoundMVResult(abs(*intlMV*)).
   b. If *rounded* is **+∞**𝔽 and *intlMV* < 0, return NEGATIVE-INFINITY.
   c. If *rounded* is **+∞**𝔽, return POSITIVE-INFINITY.
   d. If *rounded* is **+0**𝔽 and *intlMV* < 0, return NEGATIVE-ZERO.
   e. If *rounded* is **+0**𝔽, return 0.
10. Return *intlMV*.

15.5.17 GetUnsignedRoundingMode (*roundingMode*, *sign*)

The abstract operation GetUnsignedRoundingMode takes arguments *roundingMode* (a String) and *sign* (NEGATIVE or POSITIVE) and returns a specification type from the Unsigned Rounding Mode column of Table 15. It returns the rounding mode that should be applied to the absolute value of a number to produce the same result as if *roundingMode*, one of the String values in the Identifier column of Table 13, were applied to the signed value of the number (negative if *sign* is NEGATIVE, or positive otherwise). It performs the following steps when called:

1. Return the specification type in the Unsigned Rounding Mode column of Table 15 for the row where the value in the Identifier column is *roundingMode* and the value in the Sign column is *sign*.

Table 15: Conversion from rounding mode to unsigned rounding mode

Identifier	Sign	Unsigned Rounding Mode
"ceil"	POSITIVE	INFINITY
	NEGATIVE	ZERO
"floor"	POSITIVE	ZERO
	NEGATIVE	INFINITY
"expand"	POSITIVE	INFINITY
	NEGATIVE	INFINITY
"trunc"	POSITIVE	ZERO
	NEGATIVE	ZERO
"halfCeil"	POSITIVE	HALF-INFINITY
	NEGATIVE	HALF-ZERO
"halfFloor"	POSITIVE	HALF-ZERO
	NEGATIVE	HALF-INFINITY
"halfExpand"	POSITIVE	HALF-INFINITY
	NEGATIVE	HALF-INFINITY
"halfTrunc"	POSITIVE	HALF-ZERO
	NEGATIVE	HALF-ZERO
"halfEven"	POSITIVE	HALF-EVEN
	NEGATIVE	HALF-EVEN

15.5.18 ApplyUnsignedRoundingMode (*x*, *r1*, *r2*, *unsignedRoundingMode*)

The abstract operation ApplyUnsignedRoundingMode takes arguments *x* (a mathematical value), *r1* (a mathematical value), *r2* (a mathematical value), and *unsignedRoundingMode* (a specification type from the Unsigned Rounding Mode column of Table 15, or **undefined**) and returns a mathematical value. It considers *x*, bracketed below by *r1* and above by *r2*, and returns either *r1* or *r2* according to *unsignedRoundingMode*. It performs the following steps when called:

1. If *x* is equal to *r1*, return *r1*.
2. Assert: *r1* < *x* < *r2*.
3. Assert: *unsignedRoundingMode* is not **undefined**.
4. If *unsignedRoundingMode* is ZERO, return *r1*.

5. If *unsignedRoundingMode* is INFINITY, return *r2*.
6. Let *d1* be *x* – *r1*.
7. Let *d2* be *r2* – *x*.
8. If *d1* < *d2*, return *r1*.
9. If *d2* < *d1*, return *r2*.
10. Assert: *d1* is equal to *d2*.
11. If *unsignedRoundingMode* is HALF-ZERO, return *r1*.
12. If *unsignedRoundingMode* is HALF-INFINITY, return *r2*.
13. Assert: *unsignedRoundingMode* is HALF-EVEN.
14. Let *cardinality* be (*r1* / (*r2* – *r1*)) modulo 2.
15. If *cardinality* is 0, return *r1*.
16. Return *r2*.

15.5.19 PartitionNumberRangePattern (*numberFormat*, *x*, *y*)

The abstract operation PartitionNumberRangePattern takes arguments *numberFormat* (an Intl.NumberFormat), *x* (an Intl mathematical value), and *y* (an Intl mathematical value) and returns either a normal completion containing a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Source]] (a String), or a throw completion. It creates the parts for a localized number range according to *x*, *y*, and the formatting options of *numberFormat*. It performs the following steps when called:

1. If *x* is NOT-A-NUMBER or *y* is NOT-A-NUMBER, throw a **RangeError** exception.
2. Let *result* be a new empty List.
3. Let *xResult* be PartitionNumberPattern(*numberFormat*, *x*).
4. Let *yResult* be PartitionNumberPattern(*numberFormat*, *y*).
5. If FormatNumeric(*numberFormat*, *x*) is equal to FormatNumeric(*numberFormat*, *y*), then
   a. Let *appxResult* be FormatApproximately(*numberFormat*, *xResult*).
   b. For each element *r* of *appxResult*, do
      i. Set *r*.[[Source]] to **"shared"**.
   c. Return *appxResult*.
6. For each element *r* of *xResult*, do
   a. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: **"startRange"** } to *result*.
7. Let *rangeSeparator* be an ILND String value used to separate two numbers.
8. Append the Record { [[Type]]: **"literal"**, [[Value]]: *rangeSeparator*, [[Source]]: **"shared"** } to *result*.
9. For each element *r* of *yResult*, do
   a. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: **"endRange"** } to *result*.
10. Return CollapseNumberRange(*result*).

15.5.20 FormatApproximately (*numberFormat*, *result*)

The abstract operation FormatApproximately takes arguments *numberFormat* (an Intl.NumberFormat) and *result* (a List of Records with fields [[Type]] (a String) and [[Value]] (a String)) and returns a List of Records with fields [[Type]] (a String) and [[Value]] (a String). It modifies *result*, which must be a List of Record values as described in PartitionNumberPattern, by adding a new Record for the approximately sign, which may depend on *numberFormat*. It performs the following steps when called:

1. Let *approximatelySign* be an ILND String value used to signify that a number is approximate.
2. If *approximatelySign* is not empty, insert a new Record { [[Type]]: **"approximatelySign"**, [[Value]]: *approximatelySign* } at an ILND index in *result*. For example, if *numberFormat* has [[Locale]] **"en-US"** and [[NumberingSystem]] **"latn"** and [[Style]] **"decimal"**, the new Record might be inserted before the first element of *result*.
3. Return *result*.

15.5.21 CollapseNumberRange (*result*)

The implementation-defined abstract operation CollapseNumberRange takes argument *result* (a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Source]] (a String)) and returns a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Source]] (a String). It modifies *result* (which must be a List of Records as constructed within PartitionNumberRangePattern) by removing redundant information and resolving internal inconsistency, and returns the resulting List. The algorithm is implementation dependent, but must not introduce ambiguity that would cause the result of Intl.NumberFormat.prototype.formatRange (*start*, *end*) with arguments List « *start1*, *end1* » to equal the result with arguments List « *start2*, *end2* » if the results for those same arguments Lists would not be equal with a trivial implementation of CollapseNumberRange that always returns *result* unmodified.

For example, an implementation may remove the Record representing a currency symbol after a range separator to convert a *results* List representing **"$3–$5"** into one representing **"$3–5"**.

An implementation may also modify Record [[Value]] fields for grammatical correctness; for example, converting a *results* List representing **"0.5 miles–1 mile"** into one representing **"0.5–1 miles"**.

Returning *result* unmodified is guaranteed to be a correct implementation of CollapseNumberRange.

15.5.22 FormatNumericRange (*numberFormat*, *x*, *y*)

The abstract operation FormatNumericRange takes arguments *numberFormat* (an Intl.NumberFormat), *x* (an Intl mathematical value), and *y* (an Intl mathematical value) and returns either a normal completion containing a String or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionNumberRangePattern(*numberFormat*, *x*, *y*).
2. Let *result* be the empty String.
3. For each element *part* of *parts*, do
   a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

15.5.23 FormatNumericRangeToParts (*numberFormat*, *x*, *y*)

The abstract operation FormatNumericRangeToParts takes arguments *numberFormat* (an Intl.NumberFormat), *x* (an Intl mathematical value), and *y* (an Intl mathematical value) and returns either a normal completion containing an Array or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionNumberRangePattern(*numberFormat*, *x*, *y*).
2. Let *result* be ! ArrayCreate(0).
3. Let *n* be 0.
4. For each element *part* of *parts*, do
   a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
   d. Perform ! CreateDataPropertyOrThrow(*O*, **"source"**, *part*.[[Source]]).
   e. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString(𝔽(*n*)), *O*).
   f. Increment *n* by 1.
5. Return *result*.

16 PluralRules Objects

16.1 The Intl.PluralRules Constructor

The PluralRules constructor is the *%Intl.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.

16.1.1 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, **"%Intl.PluralRules.prototype%"**, « [[InitializedPluralRules]], [[Locale]], [[Type]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[RoundingType]], [[RoundingIncrement]], [[RoundingMode]], [[ComputedRoundingPriority]], [[TrailingZeroDisplay]] »).
3. Return ? InitializePluralRules(*pluralRules*, *locales*, *options*).

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

The abstract operation InitializePluralRules takes arguments *pluralRules* (an Intl.PluralRules), *locales* (an ECMAScript language value), and *options* (an ECMAScript language value) and returns either a normal completion containing *pluralRules* or a throw completion. It initializes *pluralRules* as a PluralRules object. It performs the following steps when called:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. Set *options* to ? CoerceOptionsToObject(*options*).
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 *t* be ? GetOption(*options*, **"type"**, `STRING`, « **"cardinal"**, **"ordinal"** », **"cardinal"**).
7. Set *pluralRules*.[[Type]] to *t*.
8. Perform ? SetNumberFormatDigitOptions(*pluralRules*, *options*, `0`, `3`, **"standard"**).
9. Let *r* be ResolveLocale(%Intl.PluralRules%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.PluralRules%.[[RelevantExtensionKeys]], %Intl.PluralRules%.[[LocaleData]]).
10. Set *pluralRules*.[[Locale]] to *r*.[[Locale]].
11. Return *pluralRules*.

16.2 Properties of the Intl.PluralRules Constructor

The Intl.PluralRules constructor has the following properties:

16.2.1 Intl.PluralRules.prototype

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

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

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

16.2.3 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 « ».

NOTE 1 Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> 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 https://cldr.unicode.org/).

16.3 Properties of the Intl.PluralRules Prototype Object

The Intl.PluralRules prototype object is itself an ordinary object. *%Intl.PluralRules.prototype%* 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.

16.3.1 Intl.PluralRules.prototype.constructor

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

16.3.2 Intl.PluralRules.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.PluralRules"**.

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

16.3.3 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. Perform ? RequireInternalSlot(*pr*, [[InitializedPluralRules]]).
3. Let *n* be ? ToNumber(*value*).
4. Return ResolvePlural(*pr*, *n*).[[PluralCategory]].

16.3.4 Intl.PluralRules.prototype.selectRange (*start*, *end*)

When the `selectRange` method is called with arguments *start* and *end*, the following steps are taken:

1. Let *pr* be the **this** value.
2. Perform ? RequireInternalSlot(*pr*, [[InitializedPluralRules]]).
3. If *start* is **undefined** or *end* is **undefined**, throw a **TypeError** exception.
4. Let *x* be ? ToNumber(*start*).
5. Let *y* be ? ToNumber(*end*).
6. Return ? ResolvePluralRange(*pr*, *x*, *y*).

16.3.5 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. Perform ? RequireInternalSlot(*pr*, [[InitializedPluralRules]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. Let *pluralCategories* be a List of Strings containing all possible results of PluralRuleSelect for the selected locale *pr*.[[Locale]].
5. For each row of Table 16, except the header row, in table order, do

   a. Let *p* be the Property value of the current row.
   b. If *p* is **"pluralCategories"**, then
      i. Let *v* be CreateArrayFromList(*pluralCategories*).
   c. Else,
      i. Let *v* be the value of *pr*'s internal slot whose name is the Internal Slot value of the current row.
   d. If *v* is not **undefined**, then
      i. If there is a Conversion value in the current row, then
         1. Assert: The Conversion value of the current row is NUMBER.
         2. Set *v* to 𝔽(*v*).
      ii. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
6. Return *options*.

Table 16: Resolved Options of PluralRules Instances

Internal Slot	Property	Conversion
[[Locale]]	**"locale"**	
[[Type]]	**"type"**	
[[MinimumIntegerDigits]]	**"minimumIntegerDigits"**	NUMBER
[[MinimumFractionDigits]]	**"minimumFractionDigits"**	NUMBER
[[MaximumFractionDigits]]	**"maximumFractionDigits"**	NUMBER
[[MinimumSignificantDigits]]	**"minimumSignificantDigits"**	NUMBER
[[MaximumSignificantDigits]]	**"maximumSignificantDigits"**	NUMBER
	"pluralCategories"	
[[RoundingIncrement]]	**"roundingIncrement"**	NUMBER
[[RoundingMode]]	**"roundingMode"**	
[[ComputedRoundingPriority]]	**"roundingPriority"**	
[[TrailingZeroDisplay]]	**"trailingZeroDisplay"**	

16.4 Properties of Intl.PluralRules Instances

Intl.PluralRules instances are ordinary objects that inherit properties from %Intl.PluralRules.prototype%.

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

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 indicating the minimum integer digits to be used.
- [[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integers 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 integers 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.
- [[RoundingType]] is one of the values FRACTION-DIGITS, SIGNIFICANT-DIGITS, MORE-PRECISION, or LESS-PRECISION, indicating which rounding strategy to use, as discussed in 15.4.
- [[ComputedRoundingPriority]] is one of the String values **"auto"**, **"morePrecision"**, or **"lessPrecision"**. It is only used in 16.3.5 to convert [[RoundingType]] back to a valid **"roundingPriority"** option.
- [[RoundingIncrement]] is an integer that evenly divides `10`, `100`, `1000`, or `10000` into tenths, fifths, quarters,

or halves. It indicates the increment at which rounding should take place relative to the calculated rounding magnitude. For example, if [[MaximumFractionDigits]] is 2 and [[RoundingIncrement]] is 5, then formatted numbers are rounded to the nearest `0.05` ("nickel rounding").
- [[RoundingMode]] is one of the String values in the Identifier column of Table 13, specifying which rounding mode to use.
- [[TrailingZeroDisplay]] is one of the String values **"auto"** or **"stripIfInteger"**, indicating whether to strip trailing zeros if the formatted number is an integer (i.e., has no non-zero fraction digit).

16.5 Abstract Operations for PluralRules Objects

16.5.1 GetOperands (*s*)

The abstract operation GetOperands takes argument *s* (a decimal String) and returns a Plural Rules Operands Record. It extracts numeric features from *s* that correspond with the operands of Unicode Technical Standard #35 Part 3 Numbers, Section 5.1.1 Operands <https://unicode.org/reports/tr35/tr35-numbers.html#Operands>. It performs the following steps when called:

1. Let *n* be ! ToNumber(*s*).
2. Assert: *n* is finite.
3. Let *dp* be StringIndexOf(*s*, **"."**, 0).
4. If *dp* = -1, then
   a. Let *intPart* be *n*.
   b. Let *fracSlice* be **""**.
5. Else,
   a. Let *intPart* be the substring of *s* from 0 to *dp*.
   b. Let *fracSlice* be the substring of *s* from *dp* + 1.
6. Let *i* be abs(! ToNumber(*intPart*)).
7. Let *fracDigitCount* be the length of *fracSlice*.
8. Let *f* be ! ToNumber(*fracSlice*).
9. Let *significantFracSlice* be the value of *fracSlice* stripped of trailing **"0"**.
10. Let *significantFracDigitCount* be the length of *significantFracSlice*.
11. Let *significantFrac* be ! ToNumber(*significantFracSlice*).
12. Return a new Plural Rules Operands Record { [[Number]]: abs(*n*), [[IntegerDigits]]: *i*, [[FractionDigits]]: *f*, [[NumberOfFractionDigits]]: *fracDigitCount*, [[FractionDigitsWithoutTrailing]]: *significantFrac*, [[NumberOfFractionDigitsWithoutTrailing]]: *significantFracDigitCount* }.

16.5.2 Plural Rules Operands Records

Each *Plural Rules Operands Record* has the fields defined in Table 17.

Table 17: Plural Rules Operands Record Fields

Field Name	Value Type	UTS #35 Operand	Description
[[Number]]	Number	n	Absolute value of the source number
[[IntegerDigits]]	Number	i	Integer part of [[Number]].
[[FractionDigits]]	Number	f	Visible fraction digits in [[Number]], *with* trailing zeroes, as an integer having [[NumberOfFractionDigits]] digits.
[[NumberOfFractionDigits]]	Number	v	Number of visible fraction digits in [[Number]], *with* trailing zeroes.

Table 17: Plural Rules Operands Record Fields *(continued)*

Field Name	Value Type	UTS #35 Operand	Description
[[FractionDigitsWithoutTrailing]]	Number	t	Visible fraction digits in [[Number]], *without* trailing zeroes, as an integer having [[NumberOfFractionDigitsWithoutTrailing]] digits.
[[NumberOfFractionDigitsWithoutTrailing]]	Number	w	Number of visible fraction digits in [[Number]], *without* trailing zeroes.

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

The implementation-defined abstract operation PluralRuleSelect takes arguments *locale* (a String), *type* (**"cardinal"** or **"ordinal"**), *n* (a finite Number), and *operands* (a Plural Rules Operands Record derived from formatting *n*) and returns **"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**. The returned String best categorizes the *operands* representation of *n* according to the rules for *locale* and *type*.

16.5.4 ResolvePlural (*pluralRules*, *n*)

The abstract operation ResolvePlural takes arguments *pluralRules* (an Intl.PluralRules) and *n* (a Number) and returns a Record with fields [[PluralCategory]] (**"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**) and [[FormattedString]] (a String). The returned Record contains two string-valued fields describing *n* according to the effective locale and the options of *pluralRules*: [[PluralCategory]] characterizing its plural category, and [[FormattedString]] containing its formatted representation. It performs the following steps when called:

1. If *n* is not a finite Number, then
   a. Let *s* be ! ToString(*n*).
   b. Return the Record { [[PluralCategory]]: **"other"**, [[FormattedString]]: *s* }.
2. Let *locale* be *pluralRules*.[[Locale]].
3. Let *type* be *pluralRules*.[[Type]].
4. Let *res* be FormatNumericToString(*pluralRules*, $\mathbb{R}$(*n*)).
5. Let *s* be *res*.[[FormattedString]].
6. Let *operands* be GetOperands(*s*).
7. Let *p* be PluralRuleSelect(*locale*, *type*, *n*, *operands*).
8. Return the Record { [[PluralCategory]]: *p*, [[FormattedString]]: *s* }.

16.5.5 PluralRuleSelectRange (*locale*, *type*, *xp*, *yp*)

The implementation-defined abstract operation PluralRuleSelectRange takes arguments *locale* (a String), *type* (**"cardinal"** or **"ordinal"**), *xp* (**"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**), and *yp* (**"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**) and returns **"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**. It performs an implementation-dependent algorithm to map the plural category String values *xp* and *yp*, respectively characterizing the start and end of a range, to a resolved String value for the plural form of the range as a whole denoted by *type* for the corresponding *locale*, or the String value **"other"**.

16.5.6 ResolvePluralRange (*pluralRules*, *x*, *y*)

The abstract operation ResolvePluralRange takes arguments *pluralRules* (an Intl.PluralRules), *x* (a Number), and *y* (a Number) and returns either a normal completion containing either **"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**, or a throw completion. The returned String value represents the plural form of the range starting from *x* and ending at *y* according to the effective locale and the options of *pluralRules*. It performs the following steps when called:

1. If *x* is **NaN** or *y* is **NaN**, throw a **RangeError** exception.
2. Let *xp* be ResolvePlural(*pluralRules*, *x*).

3. Let *yp* be ResolvePlural(*pluralRules*, *y*).
4. If *xp*.[[FormattedString]] is *yp*.[[FormattedString]], then
   a. Return *xp*.[[PluralCategory]].
5. Let *locale* be *pluralRules*.[[Locale]].
6. Let *type* be *pluralRules*.[[Type]].
7. Return PluralRuleSelectRange(*locale*, *type*, *xp*.[[PluralCategory]], *yp*.[[PluralCategory]]).

17 RelativeTimeFormat Objects

17.1 The Intl.RelativeTimeFormat Constructor

The RelativeTimeFormat constructor is the *%Intl.RelativeTimeFormat%* 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.

17.1.1 Intl.RelativeTimeFormat ([*locales* [, *options*]])

When the `Intl.RelativeTimeFormat` 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 *relativeTimeFormat* be ? OrdinaryCreateFromConstructor(NewTarget, **"%Intl.RelativeTimeFormat.prototype%"**, « [[InitializedRelativeTimeFormat]], [[Locale]], [[LocaleData]], [[Style]], [[Numeric]], [[NumberFormat]], [[NumberingSystem]], [[PluralRules]] »).
3. Return ? InitializeRelativeTimeFormat(*relativeTimeFormat*, *locales*, *options*).

17.1.2 InitializeRelativeTimeFormat (*relativeTimeFormat*, *locales*, *options*)

The abstract operation InitializeRelativeTimeFormat takes arguments *relativeTimeFormat* (an Intl.RelativeTime-Format), *locales* (an ECMAScript language value), and *options* (an ECMAScript language value) and returns either a normal completion containing *relativeTimeFormat* or a throw completion. It initializes *relativeTimeFormat* as a RelativeTimeFormat object. It performs the following steps when called:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. Set *options* to ? CoerceOptionsToObject(*options*).
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 *numberingSystem* be ? GetOption(*options*, **"numberingSystem"**, STRING, EMPTY, **undefined**).
7. If *numberingSystem* is not **undefined**, then
   a. If *numberingSystem* cannot be matched by the **`type`** Unicode locale nonterminal, throw a **RangeError** exception.
8. Set *opt*.[[nu]] to *numberingSystem*.
9. Let *r* be ResolveLocale(%Intl.RelativeTimeFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.RelativeTimeFormat%.[[RelevantExtensionKeys]], %Intl.RelativeTimeFormat%.[[LocaleData]]).
10. Let *locale* be *r*.[[Locale]].
11. Set *relativeTimeFormat*.[[Locale]] to *locale*.
12. Set *relativeTimeFormat*.[[LocaleData]] to *r*.[[LocaleData]].
13. Set *relativeTimeFormat*.[[NumberingSystem]] to *r*.[[nu]].
14. Let *style* be ? GetOption(*options*, **"style"**, STRING, « **"long"**, **"short"**, **"narrow"** », **"long"**).
15. Set *relativeTimeFormat*.[[Style]] to *style*.
16. Let *numeric* be ? GetOption(*options*, **"numeric"**, STRING, « **"always"**, **"auto"** », **"always"**).
17. Set *relativeTimeFormat*.[[Numeric]] to *numeric*.
18. Let *relativeTimeFormat*.[[NumberFormat]] be ! Construct(%Intl.NumberFormat%, « *locale* »).
19. Let *relativeTimeFormat*.[[PluralRules]] be ! Construct(%Intl.PluralRules%, « *locale* »).
20. Return *relativeTimeFormat*.

17.2 Properties of the Intl.RelativeTimeFormat Constructor

The Intl.RelativeTimeFormat constructor has the following properties:

17.2.1 Intl.RelativeTimeFormat.prototype

The value of **`Intl.RelativeTimeFormat.prototype`** is %Intl.RelativeTimeFormat.prototype%.

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

17.2.2 Intl.RelativeTimeFormat.supportedLocalesOf (*locales* [, *options*])

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

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

17.2.3 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 « **"nu"** ».

> NOTE 1 Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> describes one locale extension key that is relevant to relative time formatting: **"nu"** for numbering system (of formatted numbers).

The value of the [[LocaleData]] internal slot is implementation-defined within the constraints described in 9.1 and the following additional constraints, for all locale values *locale*:

- [[LocaleData]].[[<*locale*>]] has fields **"second"**, **"minute"**, **"hour"**, **"day"**, **"week"**, **"month"**, **"quarter"**, and **"year"**. Additional fields may exist with the previous names concatenated with the strings **"-narrow"** or **"-short"**. The values corresponding to these fields are Records which contain these two categories of fields:
  - **"future"** and **"past"** fields, which are Records with a field for each of the plural categories relevant for *locale*. The value corresponding to those fields is a pattern which may contain **"{0}"** to be replaced by a formatted number.
  - Optionally, additional fields whose key is the result of ToString of a Number, and whose values are literal Strings which are not treated as templates.
- 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"**.

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

17.3 Properties of the Intl.RelativeTimeFormat Prototype Object

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

17.3.1 Intl.RelativeTimeFormat.prototype.constructor

The initial value of **`Intl.RelativeTimeFormat.prototype.constructor`** is %Intl.RelativeTimeFormat%.

17.3.2 Intl.RelativeTimeFormat.prototype[@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.RelativeTimeFormat"**.

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

17.3.3 Intl.RelativeTimeFormat.prototype.format (*value*, *unit*)

When the **format** method is called with arguments *value* and *unit*, the following steps are taken:

1. Let *relativeTimeFormat* be the **this** value.
2. Perform ? RequireInternalSlot(*relativeTimeFormat*, [[InitializedRelativeTimeFormat]]).
3. Let *value* be ? ToNumber(*value*).
4. Let *unit* be ? ToString(*unit*).
5. Return ? FormatRelativeTime(*relativeTimeFormat*, *value*, *unit*).

17.3.4 Intl.RelativeTimeFormat.prototype.formatToParts (*value*, *unit*)

When the **formatToParts** method is called with arguments *value* and *unit*, the following steps are taken:

1. Let *relativeTimeFormat* be the **this** value.
2. Perform ? RequireInternalSlot(*relativeTimeFormat*, [[InitializedRelativeTimeFormat]]).
3. Let *value* be ? ToNumber(*value*).
4. Let *unit* be ? ToString(*unit*).
5. Return ? FormatRelativeTimeToParts(*relativeTimeFormat*, *value*, *unit*).

17.3.5 Intl.RelativeTimeFormat.prototype.resolvedOptions ( )

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

1. Let *relativeTimeFormat* be the **this** value.
2. Perform ? RequireInternalSlot(*relativeTimeFormat*, [[InitializedRelativeTimeFormat]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 18, 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 *relativeTimeFormat*'s internal slot whose name is the Internal Slot value of the current row.
   c. Assert: *v* is not **undefined**.
   d. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
5. Return *options*.

Table 18: Resolved Options of RelativeTimeFormat Instances

Internal Slot	Property
[[Locale]]	**"locale"**
[[Style]]	**"style"**
[[Numeric]]	**"numeric"**
[[NumberingSystem]]	**"numberingSystem"**

17.4 Properties of Intl.RelativeTimeFormat Instances

Intl.RelativeTimeFormat instances are ordinary objects that inherit properties from %Intl.RelativeTimeFormat.prototype%.

Intl.RelativeTimeFormat instances have an [[InitializedRelativeTimeFormat]] internal slot.

Intl.RelativeTimeFormat 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.
- [[LocaleData]] is a Record representing the data available to the implementation for formatting. It is the value of an entry in %Intl.RelativeTimeFormat%.[[LocaleData]] associated with either the value of [[Locale]] or a prefix thereof.
- [[Style]] is one of the String values **"long"**, **"short"**, or **"narrow"**, identifying the relative time format style used.
- [[Numeric]] is one of the String values **"always"** or **"auto"**, identifying whether numerical descriptions are always used, or used only when no more specific version is available (e.g., "1 day ago" vs "yesterday").
- [[NumberFormat]] is an Intl.NumberFormat object used for formatting.
- [[NumberingSystem]] is a String value with the **"type"** given in Unicode Technical Standard #35 Part 1 Core, Section 3.6.1 Key and Type Definitions <https://unicode.org/reports/tr35/#Key_And_Type_Definitions_> for the numbering system used for formatting.
- [[PluralRules]] is an Intl.PluralRules object used for formatting.

17.5 Abstract Operations for RelativeTimeFormat Objects

17.5.1 SingularRelativeTimeUnit (*unit*)

The abstract operation SingularRelativeTimeUnit takes argument *unit* (a String) and returns either a normal completion containing a String or a throw completion. It performs the following steps when called:

1. If *unit* is **"seconds"**, return **"second"**.
2. If *unit* is **"minutes"**, return **"minute"**.
3. If *unit* is **"hours"**, return **"hour"**.
4. If *unit* is **"days"**, return **"day"**.
5. If *unit* is **"weeks"**, return **"week"**.
6. If *unit* is **"months"**, return **"month"**.
7. If *unit* is **"quarters"**, return **"quarter"**.
8. If *unit* is **"years"**, return **"year"**.
9. If *unit* is not one of **"second"**, **"minute"**, **"hour"**, **"day"**, **"week"**, **"month"**, **"quarter"**, or **"year"**, throw a **RangeError** exception.
10. Return *unit*.

17.5.2 PartitionRelativeTimePattern (*relativeTimeFormat*, *value*, *unit*)

The abstract operation PartitionRelativeTimePattern takes arguments *relativeTimeFormat* (an Intl.RelativeTimeFormat), *value* (a Number), and *unit* (a String) and returns either a normal completion containing a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Unit]] (a String or EMPTY), or a throw completion. The returned List represents *value* according to the effective locale and the formatting options of *relativeTimeFormat*. It performs the following steps when called:

1. If *value* is **NaN**, **+∞**$_{\mathbb{F}}$, or **-∞**$_{\mathbb{F}}$, throw a **RangeError** exception.
2. Let *unit* be ? SingularRelativeTimeUnit(*unit*).
3. Let *fields* be *relativeTimeFormat*.[[LocaleData]].
4. Let *style* be *relativeTimeFormat*.[[Style]].
5. If *style* is equal to **"short"**, then
   a. Let *entry* be the string-concatenation of *unit* and **"-short"**.
6. Else if *style* is equal to **"narrow"**, then
   a. Let *entry* be the string-concatenation of *unit* and **"-narrow"**.
7. Else,
   a. Let *entry* be *unit*.
8. If *fields* doesn't have a field [[<*entry*>]], then
   a. Let *entry* be *unit*.
9. Let *patterns* be *fields*.[[<*entry*>]].
10. Let *numeric* be *relativeTimeFormat*.[[Numeric]].

11. If *numeric* is equal to **"auto"**, then
    a. Let *valueString* be ! ToString(*value*).
    b. If *patterns* has a field [[<*valueString*>]], then
        i. Let *result* be *patterns*.[[<*valueString*>]].
        ii. Return a List containing the Record { [[Type]]: **"literal"**, [[Value]]: *result* }.
12. If *value* is **-0**$_{\mathbb{F}}$ or *value* < **-0**$_{\mathbb{F}}$, then
    a. Let *tl* be **"past"**.
13. Else,
    a. Let *tl* be **"future"**.
14. Let *po* be *patterns*.[[<*tl*>]].
15. Let *fv* be PartitionNumberPattern(*relativeTimeFormat*.[[NumberFormat]], $\mathbb{R}$(*value*)).
16. Let *pr* be ResolvePlural(*relativeTimeFormat*.[[PluralRules]], *value*).[[PluralCategory]].
17. Let *pattern* be *po*.[[<*pr*>]].
18. Return MakePartsList(*pattern*, *unit*, *fv*).

17.5.3 MakePartsList (*pattern*, *unit*, *parts*)

The abstract operation MakePartsList takes arguments *pattern* (a pattern String), *unit* (a String), and *parts* (a List of Records representing a formatted Number) and returns a List of Records with fields [[Type]] (a String), [[Value]] (a String), and [[Unit]] (a String or EMPTY). It performs the following steps when called:

1. Let *patternParts* be PartitionPattern(*pattern*).
2. Let *result* be a new empty List.
3. For each Record { [[Type]], [[Value]] } *patternPart* of *patternParts*, do
    a. If *patternPart*.[[Type]] is **"literal"**, then
        i. Append the Record { [[Type]]: **"literal"**, [[Value]]: *patternPart*.[[Value]], [[Unit]]: EMPTY } to *result*.
    b. Else,
        i. Assert: *patternPart*.[[Type]] is **"0"**.
        ii. For each Record { [[Type]], [[Value]] } *part* of *parts*, do
            1. Append the Record { [[Type]]: *part*.[[Type]], [[Value]]: *part*.[[Value]], [[Unit]]: *unit* } to *result*.
4. Return *result*.

NOTE Example:

1. Return MakePartsList(**"AA{0}BB"**, **"hour"**, « Record { [[Type]]: **"integer"**, [[Value]]: **"15"** } »).

will return a List of Records like

«
{ [[Type]]: **"literal"**, [[Value]]: **"AA"**, [[Unit]]: EMPTY},
{ [[Type]]: **"integer"**, [[Value]]: **"15"**, [[Unit]]: **"hour"**},
{ [[Type]]: **"literal"**, [[Value]]: **"BB"**, [[Unit]]: EMPTY}
»

17.5.4 FormatRelativeTime (*relativeTimeFormat*, *value*, *unit*)

The abstract operation FormatRelativeTime takes arguments *relativeTimeFormat* (an Intl.RelativeTimeFormat), *value* (a Number), and *unit* (a String) and returns either a normal completion containing a String or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionRelativeTimePattern(*relativeTimeFormat*, *value*, *unit*).
2. Let *result* be an empty String.
3. For each Record { [[Type]], [[Value]], [[Unit]] } *part* of *parts*, do
    a. Set *result* to the string-concatenation of *result* and *part*.[[Value]].
4. Return *result*.

17.5.5 FormatRelativeTimeToParts (*relativeTimeFormat*, *value*, *unit*)

The abstract operation FormatRelativeTimeToParts takes arguments *relativeTimeFormat* (an Intl.RelativeTimeFormat), *value* (a Number), and *unit* (a String) and returns either a normal completion containing an Array or a throw completion. It performs the following steps when called:

1. Let *parts* be ? PartitionRelativeTimePattern(*relativeTimeFormat*, *value*, *unit*).
2. Let *result* be ! ArrayCreate(`0`).
3. Let *n* be `0`.
4. For each Record { [[Type]], [[Value]], [[Unit]] } *part* of *parts*, do
   a. Let *O* be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(*O*, **"type"**, *part*.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(*O*, **"value"**, *part*.[[Value]]).
   d. If *part*.[[Unit]] is not EMPTY, then
      i. Perform ! CreateDataPropertyOrThrow(*O*, **"unit"**, *part*.[[Unit]]).
   e. Perform ! CreateDataPropertyOrThrow(*result*, ! ToString(𝔽(*n*)), *O*).
   f. Increment *n* by 1.
5. Return *result*.

18 Segmenter Objects

18.1 The Intl.Segmenter Constructor

The Segmenter constructor is the *%Intl.Segmenter%* 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.

18.1.1 Intl.Segmenter ([*locales* [, *options*]])

When the **`Intl.Segmenter`** 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 *internalSlotsList* be « [[InitializedSegmenter]], [[Locale]], [[SegmenterGranularity]] ».
3. Let *segmenter* be ? OrdinaryCreateFromConstructor(NewTarget, **"%Intl.Segmenter.prototype%"**, *internalSlotsList*).
4. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
5. Set *options* to ? GetOptionsObject(*options*).
6. Let *opt* be a new Record.
7. Let *matcher* be ? GetOption(*options*, **"localeMatcher"**, STRING, « **"lookup"**, **"best fit"** », **"best fit"**).
8. Set *opt*.[[localeMatcher]] to *matcher*.
9. Let *r* be ResolveLocale(%Intl.Segmenter%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.Segmenter%.[[RelevantExtensionKeys]], %Intl.Segmenter%.[[LocaleData]]).
10. Set *segmenter*.[[Locale]] to *r*.[[Locale]].
11. Let *granularity* be ? GetOption(*options*, **"granularity"**, STRING, « **"grapheme"**, **"word"**, **"sentence"** », **"grapheme"**).
12. Set *segmenter*.[[SegmenterGranularity]] to *granularity*.
13. Return *segmenter*.

18.2 Properties of the Intl.Segmenter Constructor

The Intl.Segmenter constructor has the following properties:

18.2.1 Intl.Segmenter.prototype

The value of **`Intl.Segmenter.prototype`** is %Intl.Segmenter.prototype%.

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

18.2.2 Intl.Segmenter.supportedLocalesOf (*locales* [, *options*])

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

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

18.2.3 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 « ».

NOTE Intl.Segmenter does not have any relevant extension keys.

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

18.3 Properties of the Intl.Segmenter Prototype Object

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

18.3.1 Intl.Segmenter.prototype.constructor

The initial value of **Intl.Segmenter.prototype.constructor** is %Intl.Segmenter%.

18.3.2 Intl.Segmenter.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Intl.Segmenter"**.

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

18.3.3 Intl.Segmenter.prototype.segment (*string*)

The **Intl.Segmenter.prototype.segment** method is called on an Intl.Segmenter instance with argument *string* to create a Segments instance for the string using the locale and options of the Intl.Segmenter instance. The following steps are taken:

1. Let *segmenter* be the **this** value.
2. Perform ? RequireInternalSlot(*segmenter*, [[InitializedSegmenter]]).
3. Let *string* be ? ToString(*string*).
4. Return CreateSegmentsObject(*segmenter*, *string*).

18.3.4 Intl.Segmenter.prototype.resolvedOptions ( )

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

1. Let *segmenter* be the **this** value.
2. Perform ? RequireInternalSlot(*segmenter*, [[InitializedSegmenter]]).
3. Let *options* be OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 19, 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 *segmenter*'s internal slot whose name is the Internal Slot value of the current row.

   c. Assert: *v* is not **undefined**.
   d. Perform ! CreateDataPropertyOrThrow(*options*, *p*, *v*).
5. Return *options*.

Table 19: Resolved Options of Segmenter Instances

Internal Slot	Property
[[Locale]]	**"locale"**
[[SegmenterGranularity]]	**"granularity"**

18.4 Properties of Intl.Segmenter Instances

Intl.Segmenter instances are ordinary objects that inherit properties from %Intl.Segmenter.prototype%.

Intl.Segmenter instances have an [[InitializedSegmenter]] internal slot.

Intl.Segmenter instances also have 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 segmentation.
- [[SegmenterGranularity]] is one of the String values **"grapheme"**, **"word"**, or **"sentence"**, identifying the kind of text element to segment.

18.5 Segments Objects

A *Segments instance* is an object that represents the segments of a specific string, subject to the locale and options of its constructing Intl.Segmenter instance.

18.5.1 CreateSegmentsObject (*segmenter*, *string*)

The abstract operation CreateSegmentsObject takes arguments *segmenter* (an Intl.Segmenter) and *string* (a String) and returns a Segments instance. The Segments instance references *segmenter* and *string*. It performs the following steps when called:

1. Let *internalSlotsList* be « [[SegmentsSegmenter]], [[SegmentsString]] ».
2. Let *segments* be OrdinaryObjectCreate(%IntlSegmentsPrototype%, *internalSlotsList*).
3. Set *segments*.[[SegmentsSegmenter]] to *segmenter*.
4. Set *segments*.[[SegmentsString]] to *string*.
5. Return *segments*.

18.5.2 The %IntlSegmentsPrototype% Object

The *%IntlSegmentsPrototype%* object:

- is the prototype of all Segments objects.
- is an ordinary object.
- has the following properties:

18.5.2.1 %IntlSegmentsPrototype%.containing (*index*)

The **`containing`** method is called on a Segments instance with argument *index* to return a Segment Data object describing the segment in the string including the code unit at the specified index according to the locale and options of the Segments instance's constructing Intl.Segmenter instance. The following steps are taken:

1. Let *segments* be the **this** value.
2. Perform ? RequireInternalSlot(*segments*, [[SegmentsSegmenter]]).
3. Let *segmenter* be *segments*.[[SegmentsSegmenter]].
4. Let *string* be *segments*.[[SegmentsString]].
5. Let *len* be the length of *string*.
6. Let *n* be ? ToIntegerOrInfinity(*index*).
7. If *n* < `0` or *n* ≥ *len*, return **undefined**.
8. Let *startIndex* be FindBoundary(*segmenter*, *string*, *n*, `BEFORE`).
9. Let *endIndex* be FindBoundary(*segmenter*, *string*, *n*, `AFTER`).
10. Return CreateSegmentDataObject(*segmenter*, *string*, *startIndex*, *endIndex*).

18.5.2.2 %IntlSegmentsPrototype% [@@iterator] ( )

The **`@@iterator`** method is called on a Segments instance to create a Segment Iterator over its string using the locale and options of its constructing Intl.Segmenter instance. The following steps are taken:

1. Let *segments* be the **this** value.
2. Perform ? RequireInternalSlot(*segments*, [[SegmentsSegmenter]]).
3. Let *segmenter* be *segments*.[[SegmentsSegmenter]].
4. Let *string* be *segments*.[[SegmentsString]].
5. Return CreateSegmentIterator(*segmenter*, *string*).

18.5.3 Properties of Segments Instances

Segments instances are ordinary objects that inherit properties from %IntlSegmentsPrototype%.

Segments instances have a [[SegmentsSegmenter]] internal slot that references the constructing Intl.Segmenter instance.

Segments instances have a [[SegmentsString]] internal slot that references the String value whose segments they expose.

18.6 Segment Iterator Objects

A *Segment Iterator* is an object that represents a particular iteration over the segments of a specific string.

18.6.1 CreateSegmentIterator (*segmenter*, *string*)

The abstract operation CreateSegmentIterator takes arguments *segmenter* (an Intl.Segmenter) and *string* (a String) and returns a Segment Iterator. The Segment Iterator iterates over *string* using the locale and options of *segmenter*. It performs the following steps when called:

1. Let *internalSlotsList* be « [[IteratingSegmenter]], [[IteratedString]], [[IteratedStringNextSegmentCodeUnitIndex]] ».
2. Let *iterator* be OrdinaryObjectCreate(%IntlSegmentIteratorPrototype%, *internalSlotsList*).
3. Set *iterator*.[[IteratingSegmenter]] to *segmenter*.
4. Set *iterator*.[[IteratedString]] to *string*.
5. Set *iterator*.[[IteratedStringNextSegmentCodeUnitIndex]] to `0`.
6. Return *iterator*.

18.6.2 The %IntlSegmentIteratorPrototype% Object

The *%IntlSegmentIteratorPrototype%* object:

- is the prototype of all Segment Iterator objects.
- is an ordinary object.
- has a [[Prototype]] internal slot whose value is the intrinsic object %Iterator.prototype%.
- has the following properties:

18.6.2.1 %IntlSegmentIteratorPrototype%.next ( )

The **`next`** method is called on a Segment Iterator instance to advance it forward one segment and return an *IteratorResult* object either describing the new segment or declaring iteration done. The following steps are taken:

1. Let *iterator* be the **this** value.
2. Perform ? RequireInternalSlot(*iterator*, [[IteratingSegmenter]]).
3. Let *segmenter* be *iterator*.[[IteratingSegmenter]].
4. Let *string* be *iterator*.[[IteratedString]].
5. Let *startIndex* be *iterator*.[[IteratedStringNextSegmentCodeUnitIndex]].
6. Let *endIndex* be FindBoundary(*segmenter*, *string*, *startIndex*, AFTER).
7. If *endIndex* is not finite, then
   a. Return CreateIterResultObject(**undefined**, **true**).
8. Set *iterator*.[[IteratedStringNextSegmentCodeUnitIndex]] to *endIndex*.
9. Let *segmentData* be CreateSegmentDataObject(*segmenter*, *string*, *startIndex*, *endIndex*).
10. Return CreateIterResultObject(*segmentData*, **false**).

18.6.2.2 %IntlSegmentIteratorPrototype% [@@toStringTag]

The initial value of the @@toStringTag property is the String value **"Segmenter String Iterator"**.

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

18.6.3 Properties of Segment Iterator Instances

Segment Iterator instances are ordinary objects that inherit properties from %SegmentIteratorPrototype%. Segment Iterator instances are initially created with the internal slots described in Table 20.

Table 20: Internal Slots of Segment Iterator Instances

Internal Slot	Description
[[IteratingSegmenter]]	The Intl.Segmenter instance used for iteration.
[[IteratedString]]	The String value being iterated upon.
[[IteratedStringNextSegmentCodeUnitIndex]]	The code unit index in the String value being iterated upon at the start of the next segment.

18.7 Segment Data Objects

A *Segment Data object* is an object that represents a particular segment from a string.

18.7.1 CreateSegmentDataObject (*segmenter*, *string*, *startIndex*, *endIndex*)

The abstract operation CreateSegmentDataObject takes arguments *segmenter* (an Intl.Segmenter), *string* (a String), *startIndex* (a non-negative integer), and *endIndex* (a non-negative integer) and returns a Segment Data object. The Segment Data object describes the segment within *string* from *segmenter* that is bounded by the indices *startIndex* and *endIndex*. It performs the following steps when called:

1. Let *len* be the length of *string*.
2. Assert: *startIndex* ≥ 0.
3. Assert: *endIndex* ≤ *len*.
4. Assert: *startIndex* < *endIndex*.
5. Let *result* be OrdinaryObjectCreate(%Object.prototype%).
6. Let *segment* be the substring of *string* from *startIndex* to *endIndex*.
7. Perform ! CreateDataPropertyOrThrow(*result*, **"segment"**, *segment*).
8. Perform ! CreateDataPropertyOrThrow(*result*, **"index"**, 𝔽(*startIndex*)).
9. Perform ! CreateDataPropertyOrThrow(*result*, **"input"**, *string*).
10. Let *granularity* be *segmenter*.[[SegmenterGranularity]].
11. If *granularity* is **"word"**, then
    a. Let *isWordLike* be a Boolean value indicating whether the *segment* in *string* is "word-like" according to locale *segmenter*.[[Locale]].
    b. Perform ! CreateDataPropertyOrThrow(*result*, **"isWordLike"**, *isWordLike*).
12. Return *result*.

NOTE Whether a segment is "word-like" is implementation-dependent, and implementations are recommended to use locale-sensitive tailorings. In general, segments consisting solely of spaces and/or punctuation (such as those terminated with "WORD_NONE" boundaries by ICU [International Components for Unicode, documented at https://unicode-org.github.io/icu-docs/]) are not considered to be "word-like".

18.8 Abstract Operations for Segmenter Objects

18.8.1 FindBoundary (*segmenter*, *string*, *startIndex*, *direction*)

The abstract operation FindBoundary takes arguments *segmenter* (an Intl.Segmenter), *string* (a String), *startIndex* (a non-negative integer), and *direction* (BEFORE or AFTER) and returns a non-negative integer or +∞. It finds a segmentation boundary between two code units in *string* in the specified *direction* from the code unit at index *startIndex* according to the locale and options of *segmenter* and returns the immediately following code unit index (which will be infinite if no such boundary exists). It performs the following steps when called:

1. Let *locale* be *segmenter*.[[Locale]].
2. Let *granularity* be *segmenter*.[[SegmenterGranularity]].
3. Let *len* be the length of *string*.
4. If *direction* is BEFORE, then
    a. Assert: *startIndex* ≥ 0.
    b. Assert: *startIndex* < *len*.
    c. Search *string* for the last segmentation boundary that is preceded by at most *startIndex* code units from the beginning, using locale *locale* and text element granularity *granularity*.
    d. If a boundary is found, return the count of code units in *string* preceding it.
    e. Return 0.
5. Assert: *direction* is AFTER.
6. If *len* is 0 or *startIndex* ≥ *len*, return +∞.
7. Search *string* for the first segmentation boundary that follows the code unit at index *startIndex*, using locale *locale* and text element granularity *granularity*.
8. If a boundary is found, return the count of code units in *string* preceding it.
9. Return *len*.

NOTE Boundary determination is implementation-dependent, but general default algorithms are specified in Unicode Standard Annex #29 <https://unicode.org/reports/tr29/>. It is recommended that implementations use locale-sensitive tailorings such as those provided by the Common Locale Data Repository (available at https://cldr.unicode.org <https://cldr.unicode.org>).

19 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 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.

19.1 Properties of the String Prototype Object

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

This definition supersedes the definition provided in es2024, 22.1.3.12.

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(%Intl.Collator%, « *locales*, *options* »).
5. Return CompareStrings(*collator*, *S*, *thatValue*).

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.

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

This definition supersedes the definition provided in es2024, 22.1.3.26.

This function interprets a String value as a sequence of code points, as described in es2024, 6.1.4. The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Return ? TransformCase(*S*, *locales*, LOWER).

NOTE 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.

19.1.2.1 TransformCase (*S*, *locales*, *targetCase*)

The abstract operation TransformCase takes arguments *S* (a String), *locales* (an ECMAScript language value), and *targetCase* (LOWER or UPPER). It interprets *S* as a sequence of UTF-16 encoded code points, as described in es2024, 6.1.4, and returns the result of implementation- and locale-dependent (ILD) transformation into *targetCase* as a new String value. It performs the following steps when called:

1. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
2. If *requestedLocales* is not an empty List, then
   a. Let *requestedLocale* be *requestedLocales*[0].
3. Else,
   a. Let *requestedLocale* be DefaultLocale().
4. Let *noExtensionsLocale* be the String value that is *requestedLocale* with any Unicode locale extension sequences removed.
5. Let *availableLocales* be an Available Locales List which includes the language tags for which the Unicode Character Database contains language-sensitive case mappings. If the implementation supports additional locale-sensitive case mappings, *availableLocales* should also include their corresponding language tags.
6. Let *match* be LookupMatchingLocaleByPrefix(*availableLocales*, *noExtensionsLocale*).
7. If *match* is not **undefined**, let *locale* be *match*.[[locale]]; else let *locale* be **"und"**.
8. Let *codePoints* be StringToCodePoints(*S*).
9. If *targetCase* is LOWER, then
   a. Let *newCodePoints* be a List whose elements are the result of a lowercase transformation of *codePoints* according to an implementation-derived algorithm using *locale* or the Unicode Default Case Conversion algorithm.
10. Else,
   a. Assert: *targetCase* is UPPER.
   b. Let *newCodePoints* be a List whose elements are the result of an uppercase transformation of *codePoints* according to an implementation-derived algorithm using *locale* or the Unicode Default Case Conversion algorithm.
11. Return CodePointsToString(*newCodePoints*).

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-sensitive tailoring defined in the file `SpecialCasing.txt` <https://unicode.org/Public/UCD/latest/ucd/SpecialCasing.txt> of the Unicode Character Database and/or CLDR and/or any other custom tailoring. Regardless of tailoring, a conforming implementation's case transformation algorithm must always yield the same result given the same input code points, locale, and target case.

NOTE The case mapping of some code points may produce multiple code points, and therefore the result may not be the same length as the input. 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()`** and **`s.toLocaleLowerCase().toLocaleUpperCase()`** is not necessarily equal to **`s.toLocaleUpperCase()`**.

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

This definition supersedes the definition provided in es2024, 22.1.3.27.

This function interprets a String value as a sequence of code points, as described in es2024, 6.1.4. The following steps are taken:

1. Let *O* be ? RequireObjectCoercible(**this** value).
2. Let *S* be ? ToString(*O*).
3. Return ? TransformCase(*S*, *locales*, UPPER).

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.

19.2 Properties of the Number Prototype Object

The following definition(s) refer to the abstract operation thisNumberValue as defined in es2024, 21.1.3.

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

This definition supersedes the definition provided in es2024, 21.1.3.4.

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(%Intl.NumberFormat%, « *locales*, *options* »).
3. Return FormatNumeric(*numberFormat*, ! ToIntlMathematicalValue(*x*)).

19.3 Properties of the BigInt Prototype Object

The following definition(s) refer to the abstract operation thisBigIntValue as defined in es2024, 21.2.3.

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

This definition supersedes the definition provided in es2024, 21.2.3.2.

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(%Intl.NumberFormat%, « *locales*, *options* »).
3. Return FormatNumeric(*numberFormat*, ℝ(*x*)).

19.4 Properties of the Date Prototype Object

The following definition(s) refer to the abstract operation thisTimeValue as defined in es2024, 21.4.4.

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

This definition supersedes the definition provided in es2024, 21.4.4.39.

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

1. Let *dateObject* be the **this** value.
2. Perform ? RequireInternalSlot(*dateObject*, [[DateValue]]).
3. Let *x* be *dateObject*.[[DateValue]].
4. If *x* is **NaN**, return **"Invalid Date"**.
5. Let *dateFormat* be ? CreateDateTimeFormat(%Intl.DateTimeFormat%, *locales*, *options*, ANY, ALL).
6. Return ! FormatDateTime(*dateFormat*, *x*).

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

This definition supersedes the definition provided in es2024, 21.4.4.38.

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

1. Let *dateObject* be the **this** value.
2. Perform ? RequireInternalSlot(*dateObject*, [[DateValue]]).
3. Let *x* be *dateObject*.[[DateValue]].
4. If *x* is **NaN**, return **"Invalid Date"**.
5. Let *dateFormat* be ? CreateDateTimeFormat(%Intl.DateTimeFormat%, *locales*, *options*, DATE, DATE).
6. Return ! FormatDateTime(*dateFormat*, *x*).

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

This definition supersedes the definition provided in es2024, `21.4.4.40`.

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

1. Let *dateObject* be the **this** value.
2. Perform ? RequireInternalSlot(*dateObject*, [[DateValue]]).
3. Let *x* be *dateObject*.[[DateValue]].
4. If *x* is **NaN**, return **"Invalid Date"**.
5. Let *timeFormat* be ? CreateDateTimeFormat(%Intl.DateTimeFormat%, *locales*, *options*, TIME, TIME).
6. Return ! FormatDateTime(*timeFormat*, *x*).

19.5 Properties of the Array Prototype Object

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

This definition supersedes the definition provided in es2024, 23.1.3.32.

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 ? LengthOfArrayLike(*array*).
3. Let *separator* be the implementation-defined list-separator String value appropriate for the host environment's current locale (such as **", "**).
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($\mathbb{F}$(*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. Set *k* to *k* + 1.
7. Return *R*.

NOTE 1 This algorithm's steps mirror the steps taken in es2024, 23.1.3.32, 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 an implementation-defined locale-sensitive separator String. This function is analogous to **`toString`** except that it is intended to yield a locale-sensitive result corresponding with conventions of the host environment's current locale.

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.

Annex A
(informative)
Implementation Dependent Behaviour

The following aspects of this specification are implementation dependent:

- In all functionality:
  - Additional values for some properties of *options* arguments (2)
  - The default locale (6.2.3)
  - The set of available locales for each constructor (9.1)
  - The LookupMatchingLocaleByBestFit algorithm (9.2.3)
- In Collator:
  - Support for the Unicode extensions keys **"kf"**, **"kn"** and the parallel options properties **"caseFirst"**, **"numeric"** (10.1.2)
  - The set of supported **"co"** key values (collations) per locale beyond a default collation (10.2.3)
  - The set of supported **"kf"** key values (case order) per locale (10.2.3)
  - The set of supported **"kn"** key values (numeric collation) per locale (10.2.3)
  - The default search sensitivity per locale (10.2.3)
  - The default ignore punctuation value per locale (10.2.3)
  - The sort order for each supported locale and options combination (10.3.3.1)
- In DateTimeFormat:
  - The BestFitFormatMatcher algorithm (11.1.2)
  - The set of supported **"ca"** key values (calendars) per locale (11.2.3)
  - The set of supported **"nu"** key values (numbering systems) per locale (11.2.3)
  - The default hourCycle setting per locale (11.2.3)
  - The set of supported date-time formats per locale beyond a core set, including the representations used for each component and the associated patterns (11.2.3)
  - Localized weekday names, era names, month names, day period names, am/pm indicators, and time zone names (11.5.7)
  - The calendric calculations used for calendars other than **"gregory"**, and adjustments for local time zones and daylight saving time (11.5.7)
  - The set of all known registered Zone and Link names of the IANA Time Zone Database and the information about their offsets from UTC and their daylight saving time rules (6.5)
- In DisplayNames:
  - The localized names (12.2.3)
- In ListFormat:
  - The patterns used for formatting values (13.2.3)
- In Locale:
  - Support for the Unicode extensions keys **"kf"**, **"kn"** and the parallel options properties **"caseFirst"**, **"numeric"** (14.1.1)
- In NumberFormat:
  - The set of supported **"nu"** key values (numbering systems) per locale (15.2.3)
  - The patterns used for formatting values as decimal, percent, currency, or unit values per locale, with or without the sign, with or without accounting format for currencies, and in standard, compact, or scientific notation (15.5.6)
  - Localized representations of **NaN** and **Infinity** (15.5.6)
  - The implementation of numbering systems not listed in Table 14 (15.5.6)
  - Localized decimal and grouping separators (15.5.6)
  - Localized plus and minus signs (15.5.6)
  - Localized digit grouping schemata (15.5.6)
  - Localized magnitude thresholds for compact notation (15.5.6)
  - Localized symbols for compact and scientific notation (15.5.6)
  - Localized narrow, short, and long currency symbols and names (15.5.6)
  - Localized narrow, short, and long unit symbols (15.5.6)
- In PluralRules:
  - List of Strings representing the possible results of plural selection and their corresponding order per locale. (16.1.2)
- In RelativeTimeFormat:

- The set of supported **"nu"** key values (numbering systems) per locale (17.2.3)
- The patterns used for formatting values (17.2.3)
- In Segmenter:
  - Boundary determination algorithms (18.8.1)
  - Classification of segments as "word-like" (18.7.1)

Annex B
(informative)
Additions and Changes That Introduce Incompatibilities with Prior Editions

- 10.1, 15.1, 11.1 In ECMA-402, 1st Edition, constructors could be used to create Intl objects from arbitrary objects. This is no longer possible in 2nd Edition.
- 11.3.3 In ECMA-402, 1st Edition, the **"length"** property of the function object *F* was set to **+0**$_{\mathbb{F}}$. In 2nd Edition, **"length"** is set to **1**$_{\mathbb{F}}$.
- 10.3.2 In ECMA-402, 7th Edition, the @@toStringTag property of **`Intl.Collator.prototype`** was set to **"Object"**. In 8th Edition, @@toStringTag is set to **"Intl.Collator"**.
- 11.3.2 In ECMA-402, 7th Edition, the @@toStringTag property of **`Intl.DateTimeFormat.prototype`** was set to **"Object"**. In 8th Edition, @@toStringTag is set to **"Intl.DateTimeFormat"**.
- 15.3.2 In ECMA-402, 7th Edition, the @@toStringTag property of **`Intl.NumberFormat.prototype`** was set to **"Object"**. In 8th Edition, @@toStringTag is set to **"Intl.NumberFormat"**.
- 16.3.2 In ECMA-402, 7th Edition, the @@toStringTag property of **`Intl.PluralRules.prototype`** was set to **"Object"**. In 8th Edition, @@toStringTag is set to **"Intl.PluralRules"**.
- 8.1.1 In ECMA-402, 7th Edition, the @@toStringTag property of **`Intl`** was not defined. In 8th Edition, @@toStringTag is set to **"Intl"**.
- 15.1 In ECMA-402, 8th Edition, the NumberFormat constructor used to throw an error when style is **"currency"** and maximumFractionDigits was set to a value lower than the default fractional digits for that currency. This behaviour was corrected in the 9th edition, and it no longer throws an error.

Colophon

This specification is authored on GitHub <https://github.com/tc39/ecma402> in a plaintext source format called Ecmarkup <https://github.com/bterlson/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 <https://github.com/rbuckton/grammarkdown> for defining syntax and Ecmarkdown <https://github.com/domenic/ecmarkdown> for authoring algorithm steps. PDF renderings of this specification are produced using PrinceXML <https://www.prince-xml.com/>.

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.

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/