



Standard ECMA-402

12th Edition / June 2025

ECMAScript[®] 2025 Internationalization API Specification

Standard

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>



is the registered trademark of Ecma International



COPYRIGHT PROTECTED DOCUMENT

Contents	page
1 Scope	1
2 Conformance	1
3 Normative References	1
4 Overview	2
4.1 Internationalization, Localization, and Globalization	2
4.2 API Overview	2
4.3 API Conventions	3
4.4 Implementation Dependencies	3
5 Notational Conventions	4
5.1 Well-Known Intrinsic Objects	4
6 Identification of Locales, Currencies, Time Zones, Measurement Units, Numbering Systems, Collations, and Calendars	5
6.1 Case Sensitivity and Case Mapping	5
6.2 Language Tags	6
6.3 Currency Codes	7
6.4 AvailableCanonicalCurrencies ()	8
6.5 Use of the IANA Time Zone Database	8
6.6 Measurement Unit Identifiers	12
6.7 Numbering System Identifiers	15
6.8 Collation Types	15
6.9 Calendar Types	15
6.10 Pattern String Types	15
7 Requirements for Standard Built-in ECMAScript Objects	16
8 The Intl Object	16
8.1 Value Properties of the Intl Object	16
8.2 Constructor Properties of the Intl Object	16
8.3 Function Properties of the Intl Object	17
9 Locale and Parameter Negotiation	18
9.1 Internal slots of Service Constructors	18
9.2 Abstract Operations	19
10 Collator Objects	25
10.1 The Intl.Collator Constructor	25
10.2 Properties of the Intl.Collator Constructor	26
10.3 Properties of the Intl.Collator Prototype Object	27
10.4 Properties of Intl.Collator Instances	30
11 DateTimeFormat Objects	30
11.1 The Intl.DateTimeFormat Constructor	30
11.2 Properties of the Intl.DateTimeFormat Constructor	33
11.3 Properties of the Intl.DateTimeFormat Prototype Object	41
11.4 Properties of Intl.DateTimeFormat Instances	44
11.5 Abstract Operations for DateTimeFormat Objects	44
12 DisplayNames Objects	53
12.1 The Intl.DisplayNames Constructor	53
12.2 Properties of the Intl.DisplayNames Constructor	54
12.3 Properties of the Intl.DisplayNames Prototype Object	55
12.4 Properties of Intl.DisplayNames Instances	56
12.5 Abstract Operations for DisplayNames Objects	56
13 DurationFormat Objects	57
13.1 The Intl.DurationFormat Constructor	57
13.2 Properties of the Intl.DurationFormat Constructor	59
13.3 Properties of the Intl.DurationFormat Prototype Object	59
13.4 Properties of Intl.DurationFormat Instances	61
13.5 Abstract Operations for DurationFormat Objects	62

14	ListFormat Objects	70
14.1	The Intl.ListFormat Constructor	70
14.2	Properties of the Intl.ListFormat Constructor	71
14.3	Properties of the Intl.ListFormat Prototype Object	72
14.4	Properties of Intl.ListFormat Instances	73
14.5	Abstract Operations for ListFormat Objects	73
15	Locale Objects	75
15.1	The Intl.Locale Constructor	75
15.2	Properties of the Intl.Locale Constructor	78
15.3	Properties of the Intl.Locale Prototype Object	78
15.4	Properties of Intl.Locale Instances	80
15.5	Abstract Operations for Locale Objects	81
16	NumberFormat Objects	82
16.1	The Intl.NumberFormat Constructor	82
16.2	Properties of the Intl.NumberFormat Constructor	85
16.3	Properties of the Intl.NumberFormat Prototype Object	87
16.4	Properties of Intl.NumberFormat Instances	89
16.5	Abstract Operations for NumberFormat Objects	91
17	PluralRules Objects	107
17.1	The Intl.PluralRules Constructor	107
17.2	Properties of the Intl.PluralRules Constructor	107
17.3	Properties of the Intl.PluralRules Prototype Object	108
17.4	Properties of Intl.PluralRules Instances	110
17.5	Abstract Operations for PluralRules Objects	110
18	RelativeTimeFormat Objects	111
18.1	The Intl.RelativeTimeFormat Constructor	111
18.2	Properties of the Intl.RelativeTimeFormat Constructor	112
18.3	Properties of the Intl.RelativeTimeFormat Prototype Object	113
18.4	Properties of Intl.RelativeTimeFormat Instances	114
18.5	Abstract Operations for RelativeTimeFormat Objects	114
19	Segmenter Objects	116
19.1	The Intl.Segmenter Constructor	116
19.2	Properties of the Intl.Segmenter Constructor	117
19.3	Properties of the Intl.Segmenter Prototype Object	117
19.4	Properties of Intl.Segmenter Instances	118
19.5	Segments Objects	118
19.6	Segment Iterator Objects	119
19.7	Segment Data Objects	120
19.8	Abstract Operations for Segmenter Objects	121
20	Locale Sensitive Functions of the ECMAScript Language Specification	122
20.1	Properties of the String Prototype Object	122
20.2	Properties of the Number Prototype Object	124
20.3	Properties of the BigInt Prototype Object	124
20.4	Properties of the Date Prototype Object	124
20.5	Properties of the Array Prototype Object	125
	Annex A (informative) Implementation Dependent Behaviour	127
	Annex B (informative) Additions and Changes That Introduce Incompatibilities with Prior Editions	129
	Annex C (informative) Colophon	131
	Software License	133

Introduction

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

The ECMAScript 2025 Internationalization API Specification (ECMA-402 12th Edition), provides key language sensitive functionality as a complement to the ECMAScript 2025 Language Specification (ECMA-262 16th 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 Čirić 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 Ecm Markup, Ecm Markdown, 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

Ben Allen, Richard Gibson, Ujjwal Sharma
ECMA-402, 12th Edition Project Editors

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) <<https://github.com/tc39/ecma402/issues>>, [File a New Issue](https://github.com/tc39/ecma402/issues/new) <<https://github.com/tc39/ecma402/issues/new>>

Pull Requests: [All Pull Requests](https://github.com/tc39/ecma402/pulls) <<https://github.com/tc39/ecma402/pulls>>, [Create a New Pull Request](https://github.com/tc39/ecma402/pulls/new) <<https://github.com/tc39/ecma402/pulls/new>>

Test Suite: [Test262](https://github.com/tc39/test262) <<https://github.com/tc39/test262>>

TC39-TG2:

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

Editors:

- [Ben Allen \(@ben-allen\)](https://github.com/ben-allen) <<https://github.com/ben-allen>>
- [Richard Gibson \(@gibson042\)](https://github.com/gibson042) <<https://github.com/gibson042>>
- [Ujjwal Sharma \(@ryzokuken\)](https://github.com/ryzokuken) <<https://github.com/ryzokuken>>

Community:

- Matrix: [#tc39:matrix.org](https://matrix.to/#/#tc39:matrix.org) <<https://matrix.to/#/#tc39:matrix.org>>
- Matrix: [#tc39-ecma402:matrix.org](https://matrix.to/#/#tc39-ecma402:matrix.org) <<https://matrix.to/#/#tc39-ecma402:matrix.org>>

Refer to the [colophon](#) for more information on how this document is created.

COPYRIGHT NOTICE

© 2025 Ecma International

By obtaining and/or copying this work, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions.

This document may be copied, published and distributed to others, and certain derivative works of it may be prepared, copied, published, and distributed, in whole or in part, provided that the above copyright notice and this Copyright License and Disclaimer are included on all such copies and derivative works. The only derivative works that are permissible under this Copyright License and Disclaimer are:

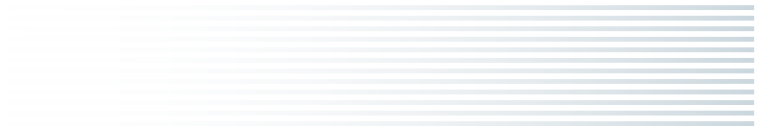
- (i) works which incorporate all or portion of this document for the purpose of providing commentary or explanation (such as an annotated version of the document),*
- (ii) works which incorporate all or portion of this document for the purpose of incorporating features that provide accessibility,*
- (iii) translations of this document into languages other than English and into different formats and*
- (iv) works by making use of this specification in standard conformant products by implementing (e.g. by copy and paste wholly or partly) the functionality therein.*

However, the content of this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as required to translate it into languages other than English or into a different format.

The official version of an Ecma International document is the English language version on the Ecma International website. In the event of discrepancies between a translated version and the official version, the official version shall govern.

The limited permissions granted above are perpetual and will not be revoked by Ecma International or its successors or assigns.

This document and the information contained herein is provided on an “AS IS” basis and ECMA INTERNATIONAL DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.



ECMAScript® 2025 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 2025 Language Specification (ECMA-262 16th 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 16** 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 2025 Language Specification (ECMA-262 16th Edition, or successor).
<https://www.ecma-international.org/publications/standards/Ecma-262.htm>

NOTE 1 Throughout this document, the phrase "es2025, x" (where x is a sequence of numbers separated by periods) may be used as shorthand for "ECMAScript 2025 Language Specification (ECMA-262 16th 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/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 2025 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 2025 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 2025 Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript 2025 Language Specification (ECMA-262 16th 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: locale selection and inspection, string comparison (collation) and case conversion, pluralization rules, text segmentation, and formatting of numbers, absolute and relative dates and times, durations, and lists. While the ECMAScript 2025 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 implementation-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 2025 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) <<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.

Throughout this specification, implementation- and locale-dependent behaviour is referred to as *ILD*, and implementation-, locale-, and numbering system-dependent behaviour is referred to as *ILND*.

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 2025 Language Specification (ECMA-262 16th Edition), as es2025:

- Object Internal Methods and Internal Slots, as described in es2025, [6.1.7.2](#).
- Algorithm conventions, as described in es2025, [5.2](#), and the use of [abstract operations](#), as described in es2025, [7.1](#), [7.2](#), [7.3](#), [7.4](#).
- Internal Slots, as described in es2025, [10.1](#).
- The [List](#) and [Record](#) Specification Type, as described in es2025, [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 es2025, [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)
%Intl.DisplayNames%	Intl.DisplayNames	The Intl.DisplayNames constructor (12.1)
%Intl.DurationFormat%	Intl.DurationFormat	The Intl.DurationFormat constructor (13.1)
%Intl.ListFormat%	Intl.ListFormat	The Intl.ListFormat constructor (14.1)
%Intl.Locale%	Intl.Locale	The Intl.Locale constructor (15.1)
%Intl.NumberFormat%	Intl.NumberFormat	The Intl.NumberFormat constructor (16.1)
%Intl.PluralRules%	Intl.PluralRules	The Intl.PluralRules constructor (17.1)
%Intl.RelativeTimeFormat%	Intl.RelativeTimeFormat	The Intl.RelativeTimeFormat constructor (18.1)
%Intl.Segmenter%	Intl.Segmenter	The Intl.Segmenter constructor (19.1)
%IntlSegmentIteratorPrototype%		The prototype of Segment Iterator objects (19.6.2)
%IntlSegmentsPrototype%		The prototype of Segments objects (19.5.2)

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, calendars, and pattern strings.

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). "I" (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) <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) <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) <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) <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) <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) <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) <https://unicode.org/reports/tr35/#BCP_47_Conformance>, return **false**.
4. Let *baseName* be `GetLocaleBaseName(lowerLocale)`.
5. Let *variants* be `GetLocaleVariants(baseName)`.
6. If *variants* is not **undefined**, then
 - a. If *variants* contains any duplicate *subtags*, return **false**.
7. Let *extensions* be the suffix of *lowerLocale* following *baseName*.
8. NOTE: A "-x-..." private use *subtag* sequence matched by the **pu_extensions** Unicode locale nonterminal must be ignored, but an isolated final "x" *subtag* with no following content does not affect any of the below checks.
9. Let *puIndex* be `StringIndexOf(extensions, "-x-", 0)`.
10. If *puIndex* is not NOT-FOUND, set *extensions* to the *substring* of *extensions* from 0 to *puIndex*.
11. 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 *tlangVariants* be [GetLocaleVariants\(tlang\)](#).
 - g. If *tlangVariants* contains any duplicate [subtags](#), return **false**.
12. 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 *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 **"-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 4217 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 sorted according to *lexicographic code unit order*, 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 Use of the IANA Time Zone Database

Implementations that adopt this specification must be *time zone aware*: they must use the IANA Time Zone Database <https://www.iana.org/time-zones/> to supply *available named time zone identifiers* and data used in ECMAScript calculations and formatting. This section defines how the IANA Time Zone Database should be used by *time zone aware* implementations. No String may be an *available named time zone identifier* unless it is a Zone name or a Link name in the IANA Time Zone Database. *Available named time zone identifiers* returned by ECMAScript built-in objects must use the casing found in the IANA Time Zone Database.

Each Zone in the IANA Time Zone Database must be a *primary time zone identifier* and each Link name in the IANA Time Zone Database must be a *non-primary time zone identifier* that resolves to its corresponding Zone name, with the following exceptions implemented in *AvailableNamedTimeZoneIdentifiers*:

- For historical reasons, **"UTC"** must be a *primary time zone identifier*. **"Etc/UTC"**, **"Etc/GMT"**, and **"GMT"**, as well as all Link names that resolve to any of them, must be non-primary time identifiers that resolve to **"UTC"**.
- Any Link name that is present in the "TZ" column of file **zone.tab** must be a *primary time zone identifier*. For example, both **"Europe/Prague"** and **"Europe/Bratislava"** must be *primary time zone identifiers*. This requirement guarantees at least one *primary time zone identifier* for each *ISO 3166-1 Alpha-2* <<https://www.iso.org/glossary-for-iso-3166.html>> country code, and ensures that future changes to time zone rules of one country will not affect ECMAScript programs that use another country's time zone(s), unless those countries' territorial boundaries have also changed.
- Any Link name that is not listed in the "TZ" column of file **zone.tab** and that represents a geographical area entirely contained within the territory of a single *ISO 3166-1 Alpha-2* <<https://www.iso.org/glossary-for-iso-3166.html>> country code must resolve to a primary identifier that also represents a geographical area entirely contained within the territory of the same country code. For example, **"Atlantic/Jan_Mayen"** must resolve to **"Arctic/Longyearbyen"**.

NOTE The IANA Time Zone Database offers build options that affect which [available named time zone identifiers](#) are primary. The default build options merge different countries' time zones, for example **"Atlantic/Reykjavik"** is built as a Link to the Zone **"Africa/Abidjan"**. Geographically and politically distinct locations are likely to introduce divergent time zone rules in a future version of the IANA Time Zone Database. The exceptions above serve to mitigate these future-compatibility issues.

The Unicode Common Locale Data Repository (CLDR <<https://cldr.unicode.org>>) implements most of the exceptions above when determining which [available named time zone identifiers](#) are primary or non-primary. Although use of CLDR data is recommended for consistency between implementations, it is not required. Non-CLDR-based implementations can still use CLDR's identifier data in **timezone.xml** <<https://github.com/unicode-org/cldr/blob/main/common/bcp47/timezone.xml>>. Implementations may also build the IANA Time Zone Database directly, for example by using build options such as **PACKRATDATA=backzone** **PACKRATLIST=zone.tab** and performing any post-processing needed to ensure compliance with the requirements above.

The IANA Time Zone Database is typically updated between five and ten times per year. These updates may add new Zone or Link names, may change Zones to Links, and may change the UTC offsets and transitions associated with any Zone. Implementations are recommended to include updates to the IANA Time Zone Database as soon as possible. Such prompt action ensures that ECMAScript programs can accurately perform time-zone-sensitive calculations and can use newly-added [available named time zone identifiers](#) supplied by external input or the [host environment](#).

Although the IANA Time Zone Database maintainers strive for stability, in rare cases (averaging less than once per year) a Zone may be replaced by a new Zone. For example, in 2022 **"Europe/Kiev"** was deprecated to a Link resolving to a new **"Europe/Kyiv"** Zone. The deprecated Link is called a *renamed time zone identifier* and the newly-added Zone is called a *replacement time zone identifier*.

To reduce disruption from these infrequent changes, implementations should initially add each replacement time zone identifier as a [non-primary time zone identifier](#) that resolves to the existing renamed time zone identifier. This allows ECMAScript programs to recognize both identifiers, but also reduces the chance that an ECMAScript program will send the replacement time zone identifier to another system that does not yet recognize it. After a *rename waiting period*, implementations should promote the new Zone to a [primary time zone identifier](#) while simultaneously demoting the renamed time zone identifier to non-primary. To provide ample time for other systems to be updated, the recommended [rename waiting period](#) is two years. However, it does not need to be either exact or dynamic. Instead, implementations should make the replacement time zone identifier primary after the waiting period as part of their normal release process for updating time zone data.

A waiting period should only apply when a new Zone is added to replace an existing Zone. If an existing Zone and Link are swapped, then no renaming has happened and no waiting period is necessary.

If implementations revise time zone information during the lifetime of an [agent](#), then it is required that the list of [available named time zone identifiers](#), the [primary time zone identifier](#) associated with any [available named time zone identifier](#), and the UTC offsets and transitions associated with any [available named time zone identifier](#), be consistent with results previously observed by that [agent](#). Due to the complexity of supporting this requirement, it is recommended that implementations maintain a fully consistent copy of the IANA Time Zone Database for the lifetime of each [agent](#).

This section complements but does not supersede [21.4.1.19](#).

6.5.1 AvailableNamedTimeZoneIdentifiers ()

The [implementation-defined](#) abstract operation AvailableNamedTimeZoneIdentifiers takes no arguments and returns a [List of Time Zone Identifier Records](#). Its result describes all [available named time zone identifiers](#) in this implementation, as well as the [primary time zone identifier](#) corresponding to each [available named time zone identifier](#). The [List](#) is ordered according to the `[[Identifier]]` field of each [Time Zone Identifier Record](#).

This definition supersedes the definition provided in [21.4.1.23](#).

1. Let *identifiers* be a [List](#) containing the String value of each Zone or Link name in the IANA Time Zone Database.
2. [Assert](#): No element of *identifiers* is an [ASCII-case-insensitive match](#) for any other element.
3. Sort *identifiers* according to [lexicographic code unit order](#).
4. Let *result* be a new empty [List](#).
5. For each element *identifier* of *identifiers*, do
 - a. Let *primary* be *identifier*.
 - b. If *identifier* is a Link name in the IANA Time Zone Database and *identifier* is not present in the "TZ" column of **zone.tab** of the IANA Time Zone Database, then
 - i. Let *zone* be the Zone name that *identifier* resolves to, according to the rules for resolving Link names in the IANA Time Zone Database.
 - ii. If *zone* starts with "Etc/", then
 1. Set *primary* to *zone*.
 - iii. Else,
 1. Let *identifierCountryCode* be the [ISO 3166-1 Alpha-2](#) <<https://www.iso.org/glossary-for-iso-3166.html>> country code whose territory contains the geographical area corresponding to *identifier*.
 2. Let *zoneCountryCode* be the ISO 3166-1 Alpha-2 country code whose territory contains the geographical area corresponding to *zone*.
 3. If *identifierCountryCode* is *zoneCountryCode*, then
 - a. Set *primary* to *zone*.
 4. Else,
 - a. Let *countryCodeLineCount* be the number of lines in file **zone.tab** of the IANA Time Zone Database where the "country-code" column is *identifierCountryCode*.
 - b. If *countryCodeLineCount* is 1, then
 - i. Let *countryCodeLine* be the line in file **zone.tab** of the IANA Time Zone Database where the "country-code" column is *identifierCountryCode*.
 - ii. Set *primary* to the contents of the "TZ" column of *countryCodeLine*.
 - c. Else,
 - i. Let *backzone* be **undefined**.
 - ii. Let *backzoneLinkLines* be the [List](#) of lines in the file **backzone** of the IANA Time Zone Database that start with either "Link " or "#PACKRATLIST zone.tab Link ".
 - iii. For each element *line* of *backzoneLinkLines*, do
 - i. Let *i* be [StringIndexOf](#)(*line*, "Link ", 0).
 - ii. Set *line* to the [substring](#) of *line* from *i* + 5.
 - iii. Let *backzoneAndLink* be [StringSplitToList](#)(*line*, " ").
 - iv. [Assert](#): *backzoneAndLink* has at least two elements, and both *backzoneAndLink*[0] and *backzoneAndLink*[1] are [available named time zone identifiers](#).
 - v. If *backzoneAndLink*[1] is *identifier*, then
 - i. [Assert](#): *backzone* is **undefined**.
 - ii. Set *backzone* to *backzoneAndLink*[0].
 - iv. [Assert](#): *backzone* is not **undefined**.
 - v. Set *primary* to *backzone*.
 - c. If *primary* is one of "Etc/UTC", "Etc/GMT", or "GMT", set *primary* to "UTC".
 - d. If *primary* is a [replacement time zone identifier](#) and its [rename waiting period](#) has not concluded, then
 - i. Let *renamedIdentifier* be the [renamed time zone identifier](#) that *primary* replaced.
 - ii. Set *primary* to *renamedIdentifier*.
 - e. Let *record* be the [Time Zone Identifier Record](#) { [[Identifier]]: *identifier*, [[PrimaryIdentifier]]: *primary* }.
 - f. Append *record* to *result*.
 6. [Assert](#): *result* contains a [Time Zone Identifier Record](#) *r* such that *r*.[[Identifier]] is "UTC" and *r*.[[PrimaryIdentifier]] is "UTC".
 7. Return *result*.

NOTE 1 The algorithm above for resolving Links to [primary time zone identifiers](#) is intended to correspond to the behaviour of `icu::TimeZone::getIanaID()` in the International Components for Unicode (ICU <<https://icu.unicode.org/>>) and the processes for maintaining [time zone identifier](#) data in the Unicode Common Locale Data Repository (CLDR <<https://cldr.unicode.org/>>).

This algorithm resolves Links to [primary time zone identifiers](#) without crossing the boundaries of ISO 3166-1 Alpha-2 country codes, using data from files **zone.tab** and **backzone** of the IANA Time Zone Database. If the country code of a Link has only one line in **zone.tab**, then that line will determine the [primary time zone identifier](#). However, if that country code has multiple lines in **zone.tab**, then historical mappings in **backzone** must be used to identify the correct [primary time zone identifier](#).

For example, to resolve "Pacific/Truk" (in country code "FM") if the default build options of the IANA Time Zone Database identify it as a Link to "Pacific/Port_Moresby" (in country code "PG"), then the "country-code" column of **zone.tab** must be checked for lines corresponding with "FM". If there were only one such line, then the "TZ" column of that line would determine the [primary time zone identifier](#) associated with "Pacific/Truk". But if there are multiple "FM" lines in **zone.tab**, then **backzone** must be inspected and a line such as "Link Pacific/Chuuk Pacific/Truk" would result in using "Pacific/Chuuk" as the [primary time zone identifier](#).

Note that **zone.tab** is the preferred source of mapping data because **backzone** mappings may, in rare cases, cross the boundaries of ISO 3166-1 Alpha-2 country codes. For example, "Atlantic/Jan_Mayen" (in country code "SJ") is mapped in **backzone** to "Europe/Oslo" (in country code "NO"). As of the 2024a release of the IANA Time Zone Database, "Atlantic/Jan_Mayen" is the only case where this happens.

NOTE 2 [Time zone identifiers](#) in the IANA Time Zone Database can change over time. At a minimum, it is recommended that implementations limit changes to the result of `AvailableNamedTimeZoneIdentifiers` to the changes allowed by `GetAvailableNamedTimeZoneIdentifier`, for the lifetime of the [surrounding agent](#). Due to the complexity of supporting these recommendations, it is recommended that the result of `AvailableNamedTimeZoneIdentifiers` remains the same for the lifetime of the [surrounding agent](#).

6.5.2 `GetAvailableNamedTimeZoneIdentifier` ([timeZoneIdentifier](#))

The abstract operation `GetAvailableNamedTimeZoneIdentifier` takes argument [timeZoneIdentifier](#) (a String) and returns either a [Time Zone Identifier Record](#) or EMPTY. If [timeZoneIdentifier](#) is an [available named time zone identifier](#), then it returns one of the [Records](#) in the [List](#) returned by `AvailableNamedTimeZoneIdentifiers`. Otherwise, EMPTY will be returned. It performs the following steps when called:

1. For each element [record](#) of `AvailableNamedTimeZoneIdentifiers()`, do
 - a. If `record.[Identifier]` is an [ASCII-case-insensitive match](#) for [timeZoneIdentifier](#), return [record](#).
2. Return EMPTY.

NOTE For any [timeZoneIdentifier](#), or any value that is an [ASCII-case-insensitive match](#) for it, it is required that the resulting [Time Zone Identifier Record](#) contain the same field values for the lifetime of the [surrounding agent](#). Furthermore, it is required that [time zone identifiers](#) not dynamically change from primary to non-primary during the lifetime of the [surrounding agent](#), meaning that if [timeZoneIdentifier](#) is an [ASCII-case-insensitive match](#) for the `[[PrimaryIdentifier]]` field of the result of a previous call to `GetAvailableNamedTimeZoneIdentifier`, then `GetAvailableNamedTimeZoneIdentifier(timeZoneIdentifier)` must return a [Time Zone Identifier Record](#) where `[[Identifier]]` is `[[PrimaryIdentifier]]`. Due to the complexity of supporting these requirements, it is recommended that the result of `AvailableNamedTimeZoneIdentifiers` (and therefore `GetAvailableNamedTimeZoneIdentifier` too) remains the same for the lifetime of the [surrounding agent](#).

6.5.3 AvailablePrimaryTimeZoneldentifiers ()

The abstract operation AvailablePrimaryTimeZoneldentifiers 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 *records* be AvailableNamedTimeZoneldentifiers().
2. Let *result* be a new empty [List](#).
3. For each element *timeZoneldentifierRecord* of *records*, do
 - a. If *timeZoneldentifierRecord*.[[Identifier]] is *timeZoneldentifierRecord*.[[PrimaryIdentifier]], then
 - i. Append *timeZoneldentifierRecord*.[[Identifier]] to *result*.
4. Return *result*.

6.5.4 StringSplitToList (*S*, *separator*)

The abstract operation StringSplitToList takes arguments *S* (a String) and *separator* (a String) and returns a [List](#) of Strings. The returned [List](#) contains all disjoint substrings of *S* that do not contain *separator* but are immediately preceded and/or immediately followed by an occurrence of *separator*. Each such substring will be the empty String between adjacent occurrences of *separator*, before a *separator* at the very start of *S*, or after a *separator* at the very end of *S*, but otherwise will not be empty. It performs the following steps when called:

1. **Assert:** *S* is not the empty String.
2. **Assert:** *separator* is not the empty String.
3. Let *separatorLength* be the length of *separator*.
4. Let *substrings* be a new empty [List](#).
5. Let *i* be 0.
6. Let *j* be StringIndexOf(*S*, *separator*, 0).
7. Repeat, while *j* is not NOT-FOUND,
 - a. Let *T* be the substring of *S* from *i* to *j*.
 - b. Append *T* to *substrings*.
 - c. Set *i* to *j* + *separatorLength*.
 - d. Set *j* to StringIndexOf(*S*, *separator*, *i*).
8. Let *T* be the substring of *S* from *i*.
9. Append *T* to *substrings*.
10. Return *substrings*.

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_Identifier) <https://unicode.org/reports/tr35/tr35-general.html#Unit_Identifier>. Their canonical form is a string containing only Unicode Basic Latin lower-case 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 NOT-FOUND or StringIndexOf(*unitIdentifier*, "-per-", *i* + 1) is not NOT-FOUND, 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) <<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://unicode.org/reports/tr35/tr35-general.html#Unit_Identifier) <https://unicode.org/reports/tr35/tr35-general.html#Unit_Identifier>, 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

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

Single Unit Identifier
kilometer
liter
megabit
megabyte
meter
microsecond
mile
mile-scandinavian
milliliter
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 sorted according to [lexicographic code unit order](#), 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) <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 sorted according to [lexicographic code unit order](#), 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 28](#), 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) <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 sorted according to [lexicographic code unit order](#), 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) <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 AvailableCalendars ()

The [implementation-defined](#) abstract operation AvailableCalendars takes no arguments and returns a [List](#) of Strings. The returned [List](#) is sorted according to [lexicographic code unit order](#), and contains unique [calendar types](#) in canonical form ([6.9](#)) identifying the calendars for which the implementation provides the functionality of Intl.DateTimeFormat objects, including their aliases (e.g., either both or neither of **"isla~~m~~icc"** and **"isla~~m~~ic-civil"**). The [List](#) must include **"iso8601"**.

6.10 Pattern String Types

Pattern String is a [String](#) value which contains zero or more substrings of the form **"{key}"**, where key can be any non-empty sequence consisting only of elements from [the ASCII word characters](#). The syntax of the abstract pattern strings is an implementation detail and is not exposed to users of ECMA-402.

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 2025 Language Specification (ECMA-262 16th 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 [[%Symbol.toStringTag%](#)]

The initial value of the [%Symbol.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.DurationFormat (. . .)

See [13](#).

8.2.5 Intl.ListFormat (. . .)

See [14](#).

8.2.6 Intl.Locale (. . .)

See 15.

8.2.7 Intl.NumberFormat (. . .)

See 16.

8.2.8 Intl.PluralRules (. . .)

See 17.

8.2.9 Intl.RelativeTimeFormat (. . .)

See 18.

8.2.10 Intl.Segmenter (. . .)

See 19.

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 a new empty [List](#).
 - b. For each element *identifier* of [AvailableCalendars\(\)](#), do
 - i. Let *canonical* be [CanonicalizeUValue\("ca", identifier\)](#).
 - ii. If *identifier* is *canonical*, then
 1. Append *identifier* to *list*.
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 [AvailablePrimaryTimeZoneIdentifiers\(\)](#).
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 *locales* is a String or *locales* is an 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 *kValue* is not a String and *kValue* is not an Object, throw a **TypeError** exception.
 - iii. If *kValue* is an 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 CanonicalizeUValue (*ukey*, *uvalue*)

The abstract operation CanonicalizeUValue takes arguments *ukey* (a Unicode locale extension sequence key 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_>) and *uvalue* (a String) and returns a String. The returned String is the canonical and case-regularized form of *uvalue* as a value of *ukey*. It performs the following steps when called:

1. Let *lowerValue* be the `ASCII-lowercase` of *uvalue*.
2. Let *canonicalized* be the String value resulting from canonicalizing *lowerValue* as a value of key *ukey* 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>>.
3. 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/>).
4. Return *canonicalized*.

9.2.3 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*.

9.2.4 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).

9.2.5 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* is NOT-FOUND, 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` }.

9.2.6 InsertUnicodeExtensionAndCanonicalize (*locale*, *attributes*, *keywords*)

The abstract operation `InsertUnicodeExtensionAndCanonicalize` takes arguments *locale* (a *language tag*), *attributes* (a *List* of Strings), and *keywords* (a *List* of `Records`) and returns a *Unicode canonicalized locale identifier*. It incorporates *attributes* and *keywords* into *locale* as a *Unicode locale extension sequence* 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 *extension* be "-u".
3. For each element *attr* of *attributes*, do
 - a. Set *extension* to the *string-concatenation* of *extension*, "-", and *attr*.
4. For each `Record` { `[[Key]]`, `[[Value]]` } *keyword* of *keywords*, do
 - a. Set *extension* to the *string-concatenation* of *extension*, "-", and *keyword*.`[[Key]]`.
 - b. If *keyword*.`[[Value]]` is not the empty String, set *extension* to the *string-concatenation* of *extension*, "-", and *keyword*.`[[Value]]`.
5. If *extension* is "-u", return `CanonicalizeUnicodeLocaleId(locale)`.
6. Let *privateIndex* be `StringIndexOf(locale, "-x-", 0)`.
7. If *privateIndex* is NOT-FOUND, then
 - a. Let *newLocale* be the *string-concatenation* of *locale* and *extension*.
8. 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*.
9. **Assert:** `IsValidLanguageTag(newLocale)` is **true**.
10. Return `CanonicalizeUnicodeLocaleId(newLocale)`.

9.2.7 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](#): *foundLocaleData* is a [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. Else,
 - a. Let *keywords* be a new empty [List](#).
12. Let *supportedKeywords* be a new empty [List](#).
13. For each element *key* of [relevantExtensionKeys](#), do
 - a. Let *keyLocaleData* be *foundLocaleData*.`[[<key>]]`.
 - b. [Assert](#): *keyLocaleData* is a [List](#).
 - c. Let *value* be *keyLocaleData*[0].
 - d. [Assert](#): *value* is a [String](#) or *value* is **null**.
 - e. Let *supportedKeyword* be **EMPTY**.
 - f. If *keywords* contains an element whose `[[Key]]` is *key*, then
 - i. Let *entry* be the element of *keywords* whose `[[Key]]` is *key*.
 - ii. Let *requestedValue* be *entry*.`[[Value]]`.
 - iii. If *requestedValue* is not the empty [String](#), then
 1. If *keyLocaleData* contains *requestedValue*, then
 - a. Set *value* to *requestedValue*.
 - b. Set *supportedKeyword* to the [Record](#) { `[[Key]]`: *key*, `[[Value]]`: *value* }.
 - iv. Else if *keyLocaleData* contains **"true"**, then
 1. Set *value* to **"true"**.
 2. Set *supportedKeyword* to the [Record](#) { `[[Key]]`: *key*, `[[Value]]`: **""** }.
 - g. [Assert](#): [options](#) has a field `[[<key>]]`.
 - h. Let *optionsValue* be [options](#).`[[<key>]]`.
 - i. [Assert](#): *optionsValue* is a [String](#), or *optionsValue* is either **undefined** or **null**.
 - j. If *optionsValue* is a [String](#), then
 - i. Let *ukey* be the [ASCII-lowercase](#) of *key*.
 - ii. Set *optionsValue* to [CanonicalizeUValue](#)(*ukey*, *optionsValue*).
 - iii. If *optionsValue* is the empty [String](#), then
 1. Set *optionsValue* to **"true"**.
 - k. If [SameValue](#)(*optionsValue*, *value*) is **false** and *keyLocaleData* contains *optionsValue*, then
 - i. Set *value* to *optionsValue*.
 - ii. Set *supportedKeyword* to **EMPTY**.
 - l. If *supportedKeyword* is not **EMPTY**, append *supportedKeyword* to *supportedKeywords*.
 - m. Set *result*.`[[<key>]]` to *value*.
 14. If *supportedKeywords* is not empty, then
 - a. Let *supportedAttributes* be a new empty [List](#).
 - b. Set *foundLocale* to [InsertUnicodeExtensionAndCanonicalize](#)(*foundLocale*, *supportedAttributes*, *supportedKeywords*).
 15. Set *result*.`[[Locale]]` to *foundLocale*.
 16. Return *result*.

9.2.8 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. If *matcher* is "lookup", then
 - i. Let *match* be [LookupMatchingLocaleByPrefix](#)(*availableLocales*, « *locale* »).
 - b. Else,
 - i. Let *match* be [LookupMatchingLocaleByBestFit](#)(*availableLocales*, « *locale* »).
 - c. If *match* is not **undefined**, append *locale* to *subset*.
5. Return [CreateArrayFromList](#)(*subset*).

9.2.9 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 *options* is an Object, then
 - a. Return *options*.
3. Throw a **TypeError** exception.

9.2.10 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*).

9.2.11 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*.

9.2.12 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*.

9.2.13 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 $\mathbb{R}(\text{value}) < \text{minimum}$ or $\mathbb{R}(\text{value}) > \text{maximum}$, throw a **RangeError** exception.
4. Return **floor**($\mathbb{R}(\text{value})$).

9.2.14 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*).

9.2.15 PartitionPattern (*pattern*)

The abstract operation **PartitionPattern** takes argument *pattern* (a **Pattern 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. It performs the following steps when called:

1. Let *result* be a new empty **List**.
2. Let *placeholderEnd* be -1.
3. Let *placeholderStart* be **StringIndexOf**(*pattern*, "{", 0).

4. Repeat, while *placeholderStart* is not NOT-FOUND,
 - a. Let *literal* be the substring of *pattern* from *placeholderEnd* + 1 to *placeholderStart*.
 - b. If *literal* is not the empty String, then
 - i. Append the Record { [[Type]]: "literal", [[Value]]: *literal* } to *result*.
 - c. Set *placeholderEnd* to StringIndexOf(*pattern*, "{", *placeholderStart*).
 - d. Assert: *placeholderEnd* is not NOT-FOUND and *placeholderStart* < *placeholderEnd*.
 - e. Let *placeholderName* be the substring of *pattern* from *placeholderStart* + 1 to *placeholderEnd*.
 - f. Append the Record { [[Type]]: *placeholderName*, [[Value]]: undefined } to *result*.
 - g. Set *placeholderStart* to StringIndexOf(*pattern*, "{", *placeholderEnd*).
5. Let *tail* be the substring of *pattern* from *placeholderEnd* + 1.
6. If *tail* is not the empty String, then
 - a. Append the Record { [[Type]]: "literal", [[Value]]: *tail* } to *result*.
7. 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. Let *requestedLocales* be ? CanonicalizeLocaleList(*locales*).
7. Set *options* to ? CoerceOptionsToObject(*options*).
8. Let *usage* be ? GetOption(*options*, "usage", STRING, « "sort", "search" », "sort").
9. Set *collator*.[[Usage]] to *usage*.
10. If *usage* is "sort", then
 - a. Let *localeData* be %Intl.Collator%.[[SortLocaleData]].
11. Else,
 - a. Let *localeData* be %Intl.Collator%.[[SearchLocaleData]].
12. Let *opt* be a new Record.
13. Let *matcher* be ? GetOption(*options*, "localeMatcher", STRING, « "lookup", "best fit" », "best fit").
14. Set *opt*.[[localeMatcher]] to *matcher*.
15. Let *collation* be ? GetOption(*options*, "collation", STRING, EMPTY, undefined).
16. If *collation* is not undefined, then
 - a. If *collation* cannot be matched by the type Unicode locale nonterminal, throw a RangeError exception.
17. Set *opt*.[[co]] to *collation*.
18. Let *numeric* be ? GetOption(*options*, "numeric", BOOLEAN, EMPTY, undefined).
19. If *numeric* is not undefined, then
 - a. Set *numeric* to ! ToString(*numeric*).
20. Set *opt*.[[kn]] to *numeric*.
21. Let *caseFirst* be ? GetOption(*options*, "caseFirst", STRING, « "upper", "lower", "false" », undefined).
22. Set *opt*.[[kf]] to *caseFirst*.
23. Let *relevantExtensionKeys* be %Intl.Collator%.[[RelevantExtensionKeys]].

24. Let *r* be `ResolveLocale(%Intl.Collator%.[[AvailableLocales]], requestedLocales, opt, relevantExtensionKeys, localeData)`.
25. Set *collator*.*[[Locale]]* to *r*.*[[Locale]]*.
26. Set *collation* to *r*.*[[co]]*.
27. If *collation* is **null**, set *collation* to **"default"**.
28. Set *collator*.*[[Collation]]* to *collation*.
29. If *relevantExtensionKeys* contains **"kn"**, then
 - a. Set *collator*.*[[Numeric]]* to `SameValue(r.[[kn]], "true")`.
30. If *relevantExtensionKeys* contains **"kf"**, then
 - a. Set *collator*.*[[CaseFirst]]* to *r*.*[[kf]]*.
31. Let *resolvedLocaleData* be *r*.*[[LocaleData]]*.
32. Let *sensitivity* be ? `GetOption(options, "sensitivity", STRING, « "base", "accent", "case", "variant" », undefined)`.
33. If *sensitivity* is **undefined**, then
 - a. If *usage* is **"sort"**, then
 - i. Set *sensitivity* to **"variant"**.
 - b. Else,
 - i. Set *sensitivity* to *resolvedLocaleData*.*[[sensitivity]]*.
34. Set *collator*.*[[Sensitivity]]* to *sensitivity*.
35. Let *defaultIgnorePunctuation* be *resolvedLocaleData*.*[[ignorePunctuation]]*.
36. Let *ignorePunctuation* be ? `GetOption(options, "ignorePunctuation", BOOLEAN, EMPTY, defaultIgnorePunctuation)`.
37. Set *collator*.*[[IgnorePunctuation]]* to *ignorePunctuation*.
38. 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) <<https://unicode.org/reports/tr35/#UnicodeCollationIdentifier>> and [Unicode Technical Standard #10 Unicode Collation Algorithm, Searching and Matching](https://unicode.org/reports/tr10/#Searching) <<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 one of the String values `"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.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 3](#), 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 3: 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"

10.3.3 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](#): *collator* is an [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_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 es2025, 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 4: 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_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 [%Symbol.toStringTag%]

The initial value of the `%Symbol.toStringTag%` property is the String value **"Intl.Collator"**.

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

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 Intl.Collator 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 representing the [Unicode Collation Identifier](#) <<https://unicode.org/reports/tr35/#UnicodeCollationIdentifier>> used for 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 3](#) 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]], [[HourCycle]], [[DateStyle]], [[TimeStyle]], [[DateTimeFormat]], [[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. Let *timeZone* be ? *Get*(*options*, "timeZone").
27. If *timeZone* is **undefined**, then
 - a. Set *timeZone* to *SystemTimeZoneIdentifier*(*options*).
28. Else,
 - a. Set *timeZone* to ? *ToString*(*timeZone*).
29. 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 × 10¹⁰).
 - f. *Assert*: *offsetMinutes* is an *integer*.
 - g. Set *timeZone* to *FormatOffsetTimeZoneIdentifier*(*offsetMinutes*).
30. Else,
 - a. Let *timeZoneIdentifierRecord* be *GetAvailableNamedTimeZoneIdentifier*(*timeZone*).
 - b. If *timeZoneIdentifierRecord* is **EMPTY**, throw a **RangeError** exception.
 - c. Set *timeZone* to *timeZoneIdentifierRecord*[[*PrimaryIdentifier*]].
31. Set *dateTimeFormat*[[*TimeZone*]] to *timeZone*.
32. Let *formatOptions* be a new *Record*.
33. Set *formatOptions*[[*hourCycle*]] to *hc*.
34. Let *hasExplicitFormatComponents* be **false**.
35. For each row of *Table 16*, 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**.
36. Let *formatMatcher* be ? *GetOption*(*options*, "formatMatcher", **STRING**, « "basic", "best fit" », "best fit").
37. Let *dateStyle* be ? *GetOption*(*options*, "dateStyle", **STRING**, « "full", "long", "medium", "short" », **undefined**).
38. Set *dateTimeFormat*[[*DateStyle*]] to *dateStyle*.
39. Let *timeStyle* be ? *GetOption*(*options*, "timeStyle", **STRING**, « "full", "long", "medium", "short" », **undefined**).
40. Set *dateTimeFormat*[[*TimeStyle*]] to *timeStyle*.
41. If *dateStyle* is not **undefined** or *timeStyle* is not **undefined**, then
 - 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*).
42. Else,
 - 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*).
43. Set *dateTimeFormat*.[[DateTimeFormat]] to *bestFormat*.
 44. If *bestFormat* has a field [[hour]], then
 - a. Set *dateTimeFormat*.[[HourCycle]] to *hc*.
 45. 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) <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 one of the String values **"h11"**, **"h12"**, **"h23"**, or **"h24"**.
- `[[LocaleData]].[<locale>].[[hourCycle12]]` must be one of the String values **"h11"** or **"h12"**.
- `[[LocaleData]].[<locale>].[[hourCycle24]]` must be one of the String values **"h23"** or **"h24"**.
- `[[LocaleData]].[<locale>]` must have a `[[formats]]` field. The value of this `[[formats]]` field must be a [Record](#) with a `[<calendar>]` field for each calendar value *calendar*. The value of each `[<calendar>]` field must be a [List](#) of [DateTime Format Records](#). Multiple [Records](#) in such a [List](#) may use the same subset of the fields as long as the corresponding values differ for at least one field. 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
 - month
 - hour, minute, second, fractionalSecondDigits
 - hour, minute, second
 - hour, minute
 - dayPeriod, hour, minute, second, fractionalSecondDigits
 - dayPeriod, hour, minute, second
 - dayPeriod, hour, minute
 - dayPeriod, hour
- `[[LocaleData]].[<locale>]` must have a `[[styles]]` field. The value of this `[[styles]]` field must be a [Record](#) with a `[<calendar>]` field for each calendar value *calendar*. The value of each `[<calendar>]` field must be a [DateTime Styles Record](#).

11.2.3.1 DateTime Format Records

Each *DateTime Format Record* has the fields defined in [Table 5](#).

Table 5: [DateTime Format Record](#)

Field Name	Value Type	Description
<code>[[weekday]]</code>	<code>[[Weekday]]</code> values in the Values column of Table 16	Optional field. Present if <code>[[pattern]]</code> contains the substring "{weekday}" .
<code>[[era]]</code>	<code>[[Era]]</code> values in the Values column of Table 16	Optional field. Present if <code>[[pattern]]</code> contains the substring "{era}" .
<code>[[year]]</code>	<code>[[Year]]</code> values in the Values column of Table 16	Optional field. Present if <code>[[pattern]]</code> contains at least one of the substrings "{year}" , "{yearName}" , or "{relatedYear}" .

Table 5: DateTime Format Record (*continued*)

Field Name	Value Type	Description
[[month]]	[[Month]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{month}".
[[day]]	[[Day]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{day}".
[[dayPeriod]]	[[DayPeriod]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{dayPeriod}".
[[hour]]	[[Hour]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{hour}".
[[minute]]	[[Minute]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{minute}".
[[second]]	[[Second]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{second}".
[[fractionalSecondDigits]]	[[FractionalSecondDigits]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{fractionalSecondDigits}".
[[timeZoneName]]	[[TimeZoneName]] values in the Values column of Table 16	Optional field. Present if [[pattern]] contains the substring "{timeZoneName}".
[[pattern]]	a Pattern String	Contains for each of the date and time format component fields of the record a substring starting with "{", followed by the name of the field, followed by "}". If the record has a [[year]] field, the string may contain the substrings "{yearName}" and "{relatedYear}".
[[pattern12]]	a Pattern String	Optional field. Present if the [[hour]] field is present. In addition to the substrings of the [[pattern]] field, contains at least one of the substrings "{ampm}" or "{dayPeriod}".
[[rangePatterns]]	a DateTime Range Pattern Record	Pattern strings in this field are similar to [[pattern]].
[[rangePatterns12]]	a DateTime Range Pattern Record	Optional field. Present if the [[hour]] field is present. Pattern strings in this field are similar to [[pattern12]].

11.2.3.2 DateTime Range Pattern Records

Each *DateTime Range Pattern Record* has the fields defined in [Table 6](#).

Table 6: DateTime Range Pattern Record

Field Name	Value Type	Description
[[Default]]	a DateTime Range Pattern Format Record	It contains the default range pattern used when a more specific range pattern is not available.
[[Era]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>era</i> is the largest calendar element that is different between the start and end dates.
[[Year]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>year</i> is the largest calendar element that is different between the start and end dates.
[[Month]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>month</i> is the largest calendar element that is different between the start and end dates.
[[Day]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>day</i> is the largest calendar element that is different between the start and end dates.
[[AmPm]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>ante</i> or <i>post meridiem</i> is the largest calendar element that is different between the start and end dates.
[[DayPeriod]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>day period</i> is the largest calendar element that is different between the start and end dates.
[[Hour]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>hour</i> is the largest calendar element that is different between the start and end dates.
[[Minute]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>minute</i> is the largest calendar element that is different between the start and end dates.
[[Second]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>second</i> is the largest calendar element that is different between the start and end dates.
[[FractionalSecondDigits]]	a DateTime Range Pattern Format Record	Optional field. Used when <i>fractional seconds</i> are the largest calendar element that is different between the start and end dates.

11.2.3.3 DateTime Range Pattern Format Records

Each *DateTime Range Pattern Format Record* has the fields defined in [Table 7](#).

Table 7: DateTime Range Pattern Format Record

Field Name	Value Type	Description
[[weekday]]	[[Weekday]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{weekday}".
[[era]]	[[Era]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{era}".
[[year]]	[[Year]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains at least one of the substrings "{year}", "{yearName}", or "{relatedYear}".
[[month]]	[[Month]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{month}".
[[day]]	[[Day]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{day}".
[[dayPeriod]]	[[DayPeriod]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{dayPeriod}".
[[hour]]	[[Hour]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{hour}".
[[minute]]	[[Minute]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{minute}".
[[second]]	[[Second]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{second}".
[[fractionalSecondDigits]]	[[FractionalSecondDigits]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{fractionalSecondDigits}".
[[timeZoneName]]	[[TimeZoneName]] values in the Values column of Table 16	Optional field. Present if a Pattern String in [[PatternParts]] contains the substring "{timeZoneName}".
[[PatternParts]]	a List of DateTime Range Pattern Part Records	Each record represents a part of the range pattern.

11.2.3.4 DateTime Range Pattern Part Records

Each *DateTime Range Pattern Part Record* has the fields defined in [Table 8](#).

Table 8: DateTime Range Pattern Part Record

Field Name	Value Type	Description
[[Source]]	"shared", "startRange", or "endRange"	It indicates which of the range's dates should be formatted using the value of the [[Pattern]] field.
[[Pattern]]	a Pattern String	A String of the same format as the regular date pattern String.

11.2.3.5 DateTime Styles Records

Each *DateTime Styles Record* has the fields defined in [Table 9](#).

Table 9: DateTime Styles Record

Field Name	Value Type
[[Date]]	a DateTime Style Record
[[Time]]	a DateTime Style Record
[[Connector]]	a DateTime Connector Record
[[DateTimeRangeFormat]]	a DateTime Date Range Record

11.2.3.6 DateTime Style Records

Each *DateTime Style Record* has the fields defined in [Table 10](#).

Table 10: DateTime Style Record

Field Name	Value Type	Description
[[full]]	a DateTime Format Record	Format record for the "full" style.
[[long]]	a DateTime Format Record	Format record for the "long" style.
[[medium]]	a DateTime Format Record	Format record for the "medium" style.
[[short]]	a DateTime Format Record	Format record for the "short" style.

11.2.3.7 DateTime Connector Records

Each *DateTime Connector Record* has the fields defined in [Table 11](#). All connector pattern strings must contain the strings "{0}" and "{1}".

Table 11: **DateTime Connector Record**

Field Name	Value Type	Description
[[full]]	a Pattern String	Connector pattern when the date style is "full".
[[long]]	a Pattern String	Connector pattern when the date style is "long".
[[medium]]	a Pattern String	Connector pattern when the date style is "medium".
[[short]]	a Pattern String	Connector pattern when the date style is "short".

11.2.3.8 DateTime Date Range Records

Each *DateTime Date Range Record* has the fields defined in [Table 12](#).

Table 12: **DateTime Date Range Record**

Field Name	Value Type	Description
[[full]]	a DateTime Time Range Record	Used when date style is "full".
[[long]]	a DateTime Time Range Record	Used when date style is "long".
[[medium]]	a DateTime Time Range Record	Used when date style is "medium".
[[short]]	a DateTime Time Range Record	Used when date style is "short".

11.2.3.9 DateTime Time Range Records

Each *DateTime Time Range Record* has the fields defined in [Table 13](#).

Table 13: **DateTime Time Range Record**

Field Name	Value Type	Description
[[full]]	a DateTime Style Range Record	Used when time style is "full".
[[long]]	a DateTime Style Range Record	Used when time style is "long".
[[medium]]	a DateTime Style Range Record	Used when time style is "medium".
[[short]]	a DateTime Style Range Record	Used when time style is "short".

11.2.3.10 DateTime Style Range Records

Each *DateTime Style Range Record* has the fields defined in [Table 14](#).

Table 14: DateTime Style Range Record

Field Name	Value Type	Description
[[rangePatterns]]	a DateTime Range Pattern Record	Range patterns to combine date and time styles.
[[rangePatterns12]]	a DateTime Range Pattern Record	Optional Field. Range patterns to combine date and time styles for 12-hour formats.

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/>).

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.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 15](#), except the header row, in table order, do
 - a. Let *p* be the Property value of the current row.
 - b. If there is an Internal Slot value in the current row, then
 - i. Let *v* be the value of *dtf*'s internal slot whose name is the Internal Slot value of the current row.
 - c. Else,
 - i. Let *format* be *dtf*.[[DateTimeFormat]].
 - ii. If *format* has a field [[<*p*>]] and *dtf*.[[DateStyle]] is **undefined** and *dtf*.[[TimeStyle]] is **undefined**, then
 1. Let *v* be *format*.[[<*p*>]].
 - iii. Else,
 1. Let *v* be **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 15: Resolved Options of DateTimeFormat Instances

Internal Slot	Property	Conversion
[[Locale]]	"locale"	
[[Calendar]]	"calendar"	
[[NumberingSystem]]	"numberingSystem"	
[[TimeZone]]	"timeZone"	
[[HourCycle]]	"hourCycle"	
[[HourCycle]]	"hour12"	HOUR12
	"weekday"	
	"era"	
	"year"	
	"month"	
	"day"	
	"dayPeriod"	
	"hour"	
	"minute"	
	"second"	
	"fractionalSecondDigits"	NUMBER
	"timeZoneName"	
[[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.

11.3.3 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 Date Time 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.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\(dtf, x, y\)](#).

11.3.5 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\(dtf, x, y\)](#).

11.3.6 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\(dtf, x\)](#).

11.3.7 Intl.DateTimeFormat.prototype [%Symbol.toStringTag%]

The initial value of the `%Symbol.toStringTag%` property is the String value **"Intl.DateTimeFormat"**.

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

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 Intl.DateTimeFormat 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 an [available named time zone identifier](#) or an [offset time zone identifier](#).
- `[[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 `[[DateTimeFormat]]` has an `[[hour]]` field.
- `[[DateStyle]]`, `[[TimeStyle]]` are each either **undefined**, or a String value with values **"full"**, **"long"**, **"medium"**, or **"short"**.
- `[[DateTimeFormat]]` is a [Date Time Format Record](#).

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 16: Components of date and time formats

Field Name	Property	Values
<code>[[Weekday]]</code>	"weekday"	"narrow" , "short" , "long"
<code>[[Era]]</code>	"era"	"narrow" , "short" , "long"
<code>[[Year]]</code>	"year"	"2-digit" , "numeric"
<code>[[Month]]</code>	"month"	"2-digit" , "numeric" , "narrow" , "short" , "long"
<code>[[Day]]</code>	"day"	"2-digit" , "numeric"
<code>[[DayPeriod]]</code>	"dayPeriod"	"narrow" , "short" , "long"
<code>[[Hour]]</code>	"hour"	"2-digit" , "numeric"
<code>[[Minute]]</code>	"minute"	"2-digit" , "numeric"
<code>[[Second]]</code>	"second"	"2-digit" , "numeric"

Table 16: Components of date and time formats (*continued*)

Field Name	Property	Values
[[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 `DateTime Styles Record`) and returns a `DateTime Format Record`. *styles* is a `Record` from `%Intl.DateTimeFormat%.
[[LocaleData]].
[[<lo-cale>]].
[[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*.
[[Time]].
[[<timeStyle>]].
3. If *dateStyle* is not **undefined**, then
 - a. **Assert:** *dateStyle* is one of "full", "long", "medium", or "short".
 - b. Let *dateFormat* be *styles*.
[[Date]].
[[<dateStyle>]].
4. If *dateStyle* is not **undefined** and *timeStyle* is not **undefined**, then
 - a. Let *format* be a new `DateTime Format 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*.
[[Connector]].
[[<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 `DateTime Format Records`) and returns a `DateTime Format 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 $-\infty$.
 9. Let *bestFormat* be **undefined**.
 10. For each element *format* of *formats*, do
 - a. Let *score* be 0.
 - b. For each row of Table 16, 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* $\neq$ *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* $\neq$ *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* $\neq$ *formatProp*, then
 - a. Set *score* to *score* - *removalPenalty*.
 - vii. Else if *optionsProp* $\neq$ *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 $\max(\min(\text{formatPropIndex} - \text{optionsPropIndex}, 2), -2)$.
 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*.
11. Return *bestFormat*.

11.5.3 BestFitFormatMatcher (*options*, *formats*)

The [implementation-defined](#) abstract operation BestFitFormatMatcher takes arguments *options* (a [Record](#)) and *formats* (a [List](#) of [DateTime Format Records](#)) and returns a [DateTime Format 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](#): *dtf* is an [Object](#) and *dtf* has an [\[\[InitializedDateTimeFormat\]\]](#) internal slot.
3. If *date* is not provided or is **undefined**, then
 - a. Let *x* be ! [Call](#)(%Date.now%, **undefined**).
4. Else,
 - a. Let *x* be ? [ToNumber](#)(*date*).
5. Return ? [FormatDateTime](#)(*dtf*, *x*).

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

11.5.5 FormatDateTimePattern (*dateTimeFormat*, *format*, *pattern*, *epochNanoseconds*)

The abstract operation FormatDateTimePattern takes arguments *dateTimeFormat* (an [Intl.DateTimeFormat](#)), *format* (a [DateTime Format Record](#) or a [DateTime Range Pattern Format Record](#)), *pattern* (a [Pattern String](#)), and *epochNanoseconds* (a [BigInt](#)) and returns a [List](#) of [Records](#) with fields [\[\[Type\]\]](#) (a [String](#)) and [\[\[Value\]\]](#) (a [String](#)). It creates the corresponding parts for the [epoch](#) time *epochNanoseconds* according to *pattern* and to the effective locale and the formatting options of *dateTimeFormat* and *format*. It performs the following steps when called:

1. Let *locale* be *dateTimeFormat*.[\[\[Locale\]\]](#).
2. Let *nfOptions* be [OrdinaryObjectCreate](#)(**null**).
3. Perform ! [CreateDataPropertyOrThrow](#)(*nfOptions*, **"numberingSystem"**, *dateTimeFormat*.[\[\[NumberingSystem\]\]](#)).
4. Perform ! [CreateDataPropertyOrThrow](#)(*nfOptions*, **"useGrouping"**, **false**).
5. Let *nf* be ! [Construct](#)(%Intl.NumberFormat%, « *locale*, *nfOptions* »).
6. Let *nf2Options* be [OrdinaryObjectCreate](#)(**null**).
7. Perform ! [CreateDataPropertyOrThrow](#)(*nf2Options*, **"minimumIntegerDigits"**, **2**_F).
8. Perform ! [CreateDataPropertyOrThrow](#)(*nf2Options*, **"numberingSystem"**, *dateTimeFormat*.[\[\[NumberingSystem\]\]](#)).
9. Perform ! [CreateDataPropertyOrThrow](#)(*nf2Options*, **"useGrouping"**, **false**).
10. Let *nf2* be ! [Construct](#)(%Intl.NumberFormat%, « *locale*, *nf2Options* »).
11. If *format* has a field [\[\[fractionalSecondDigits\]\]](#), then
 - a. Let *fractionalSecondDigits* be *format*.[\[\[fractionalSecondDigits\]\]](#).
 - b. Let *nf3Options* be [OrdinaryObjectCreate](#)(**null**).
 - c. Perform ! [CreateDataPropertyOrThrow](#)(*nf3Options*, **"minimumIntegerDigits"**, **1**_F(*fractionalSecondDigits*)).
 - d. Perform ! [CreateDataPropertyOrThrow](#)(*nf3Options*, **"numberingSystem"**, *dateTimeFormat*.[\[\[NumberingSystem\]\]](#)).
 - e. Perform ! [CreateDataPropertyOrThrow](#)(*nf3Options*, **"useGrouping"**, **false**).
 - f. Let *nf3* be ! [Construct](#)(%Intl.NumberFormat%, « *locale*, *nf3Options* »).
12. Let *tm* be [ToLocalTime](#)(*epochNanoseconds*, *dateTimeFormat*.[\[\[Calendar\]\]](#), *dateTimeFormat*.[\[\[TimeZone\]\]](#)).
13. Let *patternParts* be [PartitionPattern](#)(*pattern*).
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 **"fractionalSecondDigits"**, then

- i. **Assert:** *format* has a field `[[fractionalSecondDigits]]`.
- 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 **"dayPeriod"**, then
 - i. **Assert:** *format* has a field `[[dayPeriod]]`.
 - ii. Let *f* be *format*.`[[dayPeriod]]`.
 - iii. 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*.
 - iv. Append the **Record** { `[[Type]]`: *p*, `[[Value]]`: *fv* } to *result*.
- e. Else if *p* is **"timeZoneName"**, then
 - i. **Assert:** *format* has a field `[[timeZoneName]]`.
 - ii. Let *f* be *format*.`[[timeZoneName]]`.
 - iii. Let *v* be *dateTimeFormat*.`[[TimeZone]]`.
 - iv. 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 *f*, then use the String value of *v* itself.
 - v. Append the **Record** { `[[Type]]`: *p*, `[[Value]]`: *fv* } to *result*.
- f. Else if *p* matches a Property column of the row in Table 16, then
 - i. **Assert:** *format* has a field `[[<p>]]`.
 - ii. Let *f* be *format*.`[[<p>]]`.
 - 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. Let *codePoints* be `StringToCodePoints(fv)`.
 3. Let *count* be the number of elements in *codePoints*.
 4. If *count* > 2, then
 - a. Let *tens* be *codePoints*[*count* - 2].
 - b. Let *ones* be *codePoints*[*count* - 1].
 - c. Set *fv* to `CodePointsToString(« tens, ones »)`.
 - 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"**, then the String value may also depend on whether *format*.`[[day]]` is present. If the implementation does not have a localized representation of *f*, then use the String value of *v* itself.
 - xi. Append the **Record** { `[[Type]]`: *p*, `[[Value]]`: *fv* } to *result*.
- g. Else if *p* is **"ampm"**, then
 - i. Let *v* be *tm*.`[[Hour]]`.
 - ii. If *v* is greater than 11, then
 1. Let *fv* be an **ILD** String value representing **"post meridiem"**.
 - iii. Else,
 1. Let *fv* be an **ILD** String value representing **"ante meridiem"**.
 - iv. Append the **Record** { `[[Type]]`: **"dayPeriod"**, `[[Value]]`: *fv* } to *result*.
- h. Else if *p* is **"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 **"yearName"**, then

- i. Let *v* be *tm*.[[YearName]].
 - ii. Let *fv* be an **ILD** 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 *epochNanoseconds* 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 es2025, 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 *x* be TimeClip(*x*).
2. If *x* is NaN, throw a **RangeError** exception.
3. Let *epochNanoseconds* be $\mathbb{Z}(\mathbb{R}(x) \times 10^6)$.
4. Let *format* be *dateTimeFormat*.[[DateTimeFormat]].
5. If *dateTimeFormat*.[[HourCycle]] is "h11" or "h12", then
 - a. Let *pattern* be *format*.[[pattern12]].
6. Else,
 - a. Let *pattern* be *format*.[[pattern]].
7. Let *result* be FormatDateTimePattern(*dateTimeFormat*, *format*, *pattern*, *epochNanoseconds*).
8. 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( $\mathbb{F}(n)$ ), 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.DateTimeFormat), 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 es2025, 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 *xEpochNanoseconds* be $\mathbb{Z}(\mathbb{R}(x) \times 10^6)$.
6. Let *yEpochNanoseconds* be $\mathbb{Z}(\mathbb{R}(y) \times 10^6)$.
7. Let *tm1* be `ToLocalTime(xEpochNanoseconds, dateTimeFormat.[[Calendar]], dateTimeFormat.[[TimeZone]])`.
8. Let *tm2* be `ToLocalTime(yEpochNanoseconds, dateTimeFormat.[[Calendar]], dateTimeFormat.[[TimeZone]])`.
9. Let *format* be `dateTimeFormat.[[DateTimeFormat]]`.
10. If `dateTimeFormat.[[HourCycle]]` is **"h11"** or **"h12"**, then
 - a. Let *pattern* be `format.[[pattern12]]`.
 - b. Let *rangePatterns* be `format.[[rangePatterns12]]`.
11. Else,
 - a. Let *pattern* be `format.[[pattern]]`.
 - b. Let *rangePatterns* be `format.[[rangePatterns]]`.
12. Let *selectedRangePattern* be **undefined**.
13. Let *relevantFieldsEqual* be **true**.
14. Let *checkMoreFields* be **true**.
15. For each row of Table 6, except the header row, in table order, do
 - a. Let *fieldName* be the name given in the Field Name column of the row.
 - b. If *rangePatterns* has a field whose name is *fieldName*, let *rangePattern* be *rangePatterns*' field whose name is *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 *fieldName* is not equal to `[[Default]]` and *relevantFieldsEqual* is **true** and *checkMoreFields* is **true**, then
 - i. Set *selectedRangePattern* to *rangePattern*.
 - ii. If *fieldName* is `[[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 `[[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 `[[FractionalSecondDigits]]`, then
 1. If *format* has a `[[fractionalSecondDigits]]` field, then
 - a. Let *fractionalSecondDigits* be `format.[[fractionalSecondDigits]]`.
 2. Else,
 - a. Let *fractionalSecondDigits* be 3.
 3. Let *exp* be *fractionalSecondDigits* - 3.
 4. Let *v1* be `floor(tm1.[[Millisecond]] × 10exp)`.
 5. Let *v2* be `floor(tm2.[[Millisecond]] × 10exp)`.

- v. Else,
 - 1. Let *v1* be *tm1*'s field whose name is *fieldName*.
 - 2. Let *v2* be *tm2*'s field whose name is *fieldName*.
- vi. If *v1* is not equal to *v2*, then
 - 1. Set *relevantFieldsEqual* to **false**.
- 16. If *relevantFieldsEqual* is **true**, then
 - a. Let *collapsedResult* be a new empty List.
 - b. Let *resultParts* be *FormatDateTimePattern*(*dateTimeFormat*, *format*, *pattern*, *xEpochNanoseconds*).
 - c. For each Record { [[Type]], [[Value]] } *r* of *resultParts*, do
 - i. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: "shared" } to *collapsedResult*.
 - d. Return *collapsedResult*.
- 17. Let *rangeResult* be a new empty List.
- 18. If *selectedRangePattern* is **undefined**, then
 - a. Set *selectedRangePattern* to *rangePatterns*.[[Default]].
- 19. 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 *xEpochNanoseconds*.
 - d. Else,
 - i. Let *z* be *yEpochNanoseconds*.
 - e. Let *resultParts* be *FormatDateTimePattern*(*dateTimeFormat*, *selectedRangePattern*, *pattern*, *z*).
 - f. For each Record { [[Type]], [[Value]] } *r* of *resultParts*, do
 - i. Append the Record { [[Type]]: *r*.[[Type]], [[Value]]: *r*.[[Value]], [[Source]]: *source* } to *rangeResult*.
- 20. 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.DateTimeFormat), *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](#): [GetAvailableNamedTimeZoneIdentifier](#)(*timeZoneIdentifier*) is not EMPTY.
 - b. Let *offsetNs* be [GetNamedTimeZoneOffsetNanoseconds](#)(*timeZoneIdentifier*, *epochNs*).
3. Let *tz* be $\mathbb{R}(\text{epochNs}) + \text{offsetNs}$.
4. If *calendar* is **"gregory"**, then
 - a. Return a [ToLocalTime Record](#) with fields calculated from *tz* according to [Table 17](#).
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*.

11.5.13 ToLocalTime Records

Each *ToLocalTime Record* has the fields defined in [Table 17](#).

Table 17: [Record](#) returned by [ToLocalTime](#)

Field Name	Value Type	Value Calculation for Gregorian Calendar
[[Weekday]]	an integer	$\mathbb{R}(\text{WeekDay}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Era]]	a String	Let <i>year</i> be YearFromTime ($\mathbb{F}(\text{floor}(tz / 10^6))$). If <i>year</i> < 1 _ℱ , return "BC" , else return "AD" .
[[Year]]	an integer	$\mathbb{R}(\text{YearFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[RelatedYear]]	an integer or undefined	undefined
[[YearName]]	a String or undefined	undefined
[[Month]]	an integer	$\mathbb{R}(\text{MonthFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Day]]	an integer	$\mathbb{R}(\text{DateFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Hour]]	an integer	$\mathbb{R}(\text{HourFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Minute]]	an integer	$\mathbb{R}(\text{MinFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Second]]	an integer	$\mathbb{R}(\text{SecFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[Millisecond]]	an integer	$\mathbb{R}(\text{msFromTime}(\mathbb{F}(\text{floor}(tz / 10^6))))$
[[InDST]]	a Boolean	Calculate true or false using the best available information about the specified <i>calendar</i> and <i>timeZoneIdentifier</i> , 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 *dtf* is not an 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 19](#). 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.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 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 *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 18: Resolved Options of DisplayNames Instances

Internal Slot	Property
<code>[[Locale]]</code>	"locale"
<code>[[Style]]</code>	"style"
<code>[[Type]]</code>	"type"
<code>[[Fallback]]</code>	"fallback"
<code>[[LanguageDisplay]]</code>	"languageDisplay"

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 [%Symbol.toStringTag%]

The initial value of the `%Symbol.toStringTag%` property is the String value `"Intl.DisplayNames"`.

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

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 Intl.DisplayNames 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 19](#), return **true**.
2. Return **false**.

Table 19: 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 DurationFormat Objects

13.1 The Intl.DurationFormat Constructor

The `DurationFormat` constructor is the `%Intl.DurationFormat%` 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.DurationFormat ([*locales* [, *options*]])

When the **Intl.DurationFormat** 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 *durationFormat* be ? `OrdinaryCreateFromConstructor`(*NewTarget*, "%Intl.DurationFormatPrototype%", « [[InitializedDurationFormat]], [[Locale]], [[NumberingSystem]], [[Style]], [[YearsOptions]], [[MonthsOptions]], [[WeeksOptions]], [[DaysOptions]], [[HoursOptions]], [[MinutesOptions]], [[SecondsOptions]],

- [[MillisecondsOptions]], [[MicrosecondsOptions]], [[NanosecondsOptions]], [[HourMinuteSeparator]], [[MinuteSecondSeparator]], [[FractionalDigits]] »).
3. Let *requestedLocales* be ? *CanonicalizeLocaleList*(*locales*).
4. Let *options* be ? *GetOptionsObject*(*options*).
5. Let *matcher* be ? *GetOption*(*options*, "localeMatcher", STRING, « "lookup", "best fit" », "best fit").
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. Let *opt* be the *Record* { [[localeMatcher]]: *matcher*, [[nu]]: *numberingSystem* }.
9. Let *r* be *ResolveLocale*(%Intl.DurationFormat%.[[AvailableLocales]], *requestedLocales*, *opt*, %Intl.DurationFormat%.[[RelevantExtensionKeys]], %Intl.DurationFormat%.[[LocaleData]]).
10. Set *durationFormat*.[[Locale]] to *r*.[[Locale]].
11. Let *resolvedLocaleData* be *r*.[[LocaleData]].
12. Let *digitalFormat* be *resolvedLocaleData*.[[DigitalFormat]].
13. Set *durationFormat*.[[HourMinuteSeparator]] to *digitalFormat*.[[HourMinuteSeparator]].
14. Set *durationFormat*.[[MinuteSecondSeparator]] to *digitalFormat*.[[MinuteSecondSeparator]].
15. Set *durationFormat*.[[NumberingSystem]] to *r*.[[nu]].
16. Let *style* be ? *GetOption*(*options*, "style", STRING, « "long", "short", "narrow", "digital" », "short").
17. Set *durationFormat*.[[Style]] to *style*.
18. Let *prevStyle* be the empty String.
19. For each row of *Table 20*, except the header row, in table order, do
 - a. Let *slot* be the Internal Slot value of the current row.
 - b. Let *unit* be the Unit value of the current row.
 - c. Let *styles* be the Styles value of the current row.
 - d. Let *digitalBase* be the Digital Default value of the current row.
 - e. Let *unitOptions* be ? *GetDurationUnitOptions*(*unit*, *options*, *style*, *styles*, *digitalBase*, *prevStyle*, *digitalFormat*.[[TwoDigitHours]]).
 - f. Set the value of *durationFormat*'s internal slot whose name is *slot* to *unitOptions*.
 - g. If *unit* is one of "hours", "minutes", "seconds", "milliseconds", or "microseconds", then
 - i. Set *prevStyle* to *unitOptions*.[[Style]].
20. Set *durationFormat*.[[FractionalDigits]] to ? *GetNumberOption*(*options*, "fractionalDigits", 0, 9, undefined).
21. Return *durationFormat*.

Table 20: Internal slots and property names of DurationFormat instances

Internal Slot	Unit	Styles	Digital Default
[[YearsOptions]]	"years"	« "long", "short", "narrow" »	"short"
[[MonthsOptions]]	"months"	« "long", "short", "narrow" »	"short"
[[WeeksOptions]]	"weeks"	« "long", "short", "narrow" »	"short"
[[DaysOptions]]	"days"	« "long", "short", "narrow" »	"short"
[[HoursOptions]]	"hours"	« "long", "short", "narrow", "numeric", "2-digit" »	"numeric"
[[MinutesOptions]]	"minutes"	« "long", "short", "narrow", "numeric", "2-digit" »	"numeric"
[[SecondsOptions]]	"seconds"	« "long", "short", "narrow", "numeric", "2-digit" »	"numeric"
[[MillisecondsOptions]]	"milliseconds"	« "long", "short", "narrow", "numeric" »	"numeric"
[[MicrosecondsOptions]]	"microseconds"	« "long", "short", "narrow", "numeric" »	"numeric"
[[NanosecondsOptions]]	"nanoseconds"	« "long", "short", "narrow", "numeric" »	"numeric"

13.2 Properties of the Intl.DurationFormat Constructor

The Intl.DurationFormat [constructor](#) has the following properties:

13.2.1 Intl.DurationFormat.prototype

The value of **Intl.DurationFormat.prototype** is [%Intl.DurationFormat.prototype%](#).

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

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

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

1. Let *availableLocales* be [%Intl.DurationFormat%.\[\[AvailableLocales\]\]](#).
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 « **"nu"** ».

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*:

- [\[\[LocaleData\]\].\[<locale>\]](#) must be a [Record](#) with fields [\[\[nu\]\]](#) and [\[\[DigitalFormat\]\]](#).
- [\[\[LocaleData\]\].\[<locale>\].\[\[nu\]\]](#) must be a [List](#) as specified in [16.2.3](#) and must not include the values **"native"**, **"traditio"**, or **"finance"**.
- [\[\[LocaleData\]\].\[<locale>\].\[\[DigitalFormat\]\]](#) must be a [Record](#) with keys corresponding to each numbering system available for *locale*. Each value associated with one of those keys must be a [Record](#) containing the following fields:
 - [\[\[HourMinuteSeparator\]\]](#) must be a String value that is the appropriate separator between hours and minutes for that combination of locale and numbering system when using style **"numeric"** or **"2-digit"**.
 - [\[\[MinuteSecondSeparator\]\]](#) must be a String value that is the appropriate separator between minutes and seconds for that combination of locale and numbering system when using style **"numeric"** or **"2-digit"**.
 - [\[\[TwoDigitHours\]\]](#) must be a Boolean value indicating whether hours are always displayed using two digits when using style **"numeric"**.

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

13.3 Properties of the Intl.DurationFormat Prototype Object

The Intl.DurationFormat prototype object is itself an [ordinary object](#). [%Intl.DurationFormat.prototype%](#) is not an Intl.DurationFormat instance and does not have an [\[\[InitializedDurationFormat\]\]](#) internal slot or any of the other internal slots of Intl.DurationFormat instance objects.

13.3.1 Intl.DurationFormat.prototype.constructor

The initial value of **Intl.DurationFormat.prototype.constructor** is the intrinsic object [%Intl.DurationFormat%](#).

13.3.2 Intl.DurationFormat.prototype.resolvedOptions ()

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

1. Let *df* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*df*, [[InitializedDurationFormat]]).
3. Let *options* be [OrdinaryObjectCreate](#)(%Object.prototype%).
4. For each row of [Table 21](#), 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 *df*'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, let *conversion* be that value; else let *conversion* be **EMPTY**.
 - ii. If *conversion* is **NUMBER**, then
 1. Set *v* to $\mathbb{F}(v)$.
 - iii. Else if *conversion* is not **EMPTY**, then
 1. **Assert**: *conversion* is **STYLE+DISPLAY** and *v* is a [Duration Unit Options Record](#).
 2. **NOTE**: *v*.[[Style]] will be represented with a property named *p* (a plural Temporal unit), then *v*.[[Display]] will be represented with a property whose name suffixes *p* with **"Display"**.
 3. Let *style* be *v*.[[Style]].
 4. If *style* is **"fractional"**, then
 - a. **Assert**: [IsFractionalSecondUnitName](#)(*p*) is **true**.
 - b. Set *style* to **"numeric"**.
 5. Perform ! [CreateDataPropertyOrThrow](#)(*options*, *p*, *style*).
 6. Set *p* to the [string-concatenation](#) of *p* and **"Display"**.
 7. Set *v* to *v*.[[Display]].
 - iv. Perform ! [CreateDataPropertyOrThrow](#)(*options*, *p*, *v*).
5. Return *options*.

Table 21: Resolved Options of DurationFormat Instances

Internal Slot	Property	Conversion
[[Locale]]	"locale"	
[[NumberingSystem]]	"numberingSystem"	
[[Style]]	"style"	
[[YearsOptions]]	"years"	STYLE+DISPLAY
[[MonthsOptions]]	"months"	STYLE+DISPLAY
[[WeeksOptions]]	"weeks"	STYLE+DISPLAY
[[DaysOptions]]	"days"	STYLE+DISPLAY
[[HoursOptions]]	"hours"	STYLE+DISPLAY
[[MinutesOptions]]	"minutes"	STYLE+DISPLAY
[[SecondsOptions]]	"seconds"	STYLE+DISPLAY
[[MillisecondsOptions]]	"milliseconds"	STYLE+DISPLAY
[[MicrosecondsOptions]]	"microseconds"	STYLE+DISPLAY
[[NanosecondsOptions]]	"nanoseconds"	STYLE+DISPLAY
[[FractionalDigits]]	"fractionalDigits"	NUMBER

13.3.3 Intl.DurationFormat.prototype.format (*duration*)

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

1. Let *df* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*df*, [[InitializedDurationFormat]]).
3. Let *record* be ? [ToDurationRecord](#)(*duration*).
4. Let *parts* be [PartitionDurationFormatPattern](#)(*df*, *record*).
5. Let *result* be the empty String.
6. For each *Record* { [[Type]], [[Value]], [[Unit]] } *part* in *parts*, do
 - a. Set *result* to the [string-concatenation](#) of *result* and *part*.[[Value]].
7. Return *result*.

13.3.4 Intl.DurationFormat.prototype.formatToParts (*duration*)

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

1. Let *df* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*df*, [[InitializedDurationFormat]]).
3. Let *record* be ? [ToDurationRecord](#)(*duration*).
4. Let *parts* be [PartitionDurationFormatPattern](#)(*df*, *record*).
5. Let *result* be ! [ArrayCreate](#)(0).
6. Let *n* be 0.
7. For each *Record* { [[Type]], [[Value]], [[Unit]] } *part* in *parts*, do
 - a. Let *obj* be [OrdinaryObjectCreate](#)(%Object.prototype%).
 - b. Perform ! [CreateDataPropertyOrThrow](#)(*obj*, "type", *part*.[[Type]]).
 - c. Perform ! [CreateDataPropertyOrThrow](#)(*obj*, "value", *part*.[[Value]]).
 - d. If *part*.[[Unit]] is not EMPTY, perform ! [CreateDataPropertyOrThrow](#)(*obj*, "unit", *part*.[[Unit]]).
 - e. Perform ! [CreateDataPropertyOrThrow](#)(*result*, ! [ToString](#)(*n*), *obj*).
 - f. Set *n* to *n* + 1.
8. Return *result*.

13.3.5 Intl.DurationFormat.prototype [%Symbol.toStringTag%]

The initial value of the %Symbol.toStringTag% property is the String value "Intl.DurationFormat".

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

13.4 Properties of Intl.DurationFormat Instances

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

Intl.DurationFormat instances have an [[InitializedDurationFormat]] internal slot.

Intl.DurationFormat instances also have several internal slots that are computed by [The Intl.DurationFormat Constructor](#):

- [[Locale]] is a String value with the [language tag](#) of the locale whose localization is used for formatting.
- [[NumberingSystem]] is a String value representing the [Unicode Number System Identifier](#) <<https://unicode.org/reports/tr35/#UnicodeNumberSystemIdentifier>> used for formatting.
- [[Style]] is one of the String values "long", "short", "narrow", or "digital" identifying the duration formatting style used.
- [[YearsOptions]] is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a [Duration Record](#)'s [[Years]] field.
- [[MonthsOptions]] is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a [Duration Record](#)'s [[Months]] field.
- [[WeeksOptions]] is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a [Duration Record](#)'s [[Weeks]] field.
- [[DaysOptions]] is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a

Duration Record's `[[Days]]` field.

- `[[HoursOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Hours]]` field.
- `[[MinutesOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Minutes]]` field.
- `[[SecondsOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Seconds]]` field.
- `[[MillisecondsOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Milliseconds]]` field.
- `[[MicrosecondsOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Microseconds]]` field.
- `[[NanosecondsOptions]]` is a [Duration Unit Options Record](#) identifying the formatting style and display criteria for a Duration Record's `[[Nanoseconds]]` field.
- `[[HourMinuteSeparator]]` is a [String](#) value identifying the separator to be used between hours and minutes when both fields are displayed and both fields are formatted using numeric styles.
- `[[MinuteSecondSeparator]]` is a [String](#) value identifying the separator to be used between minutes and seconds when both fields are displayed and both fields are formatted using numeric styles.
- `[[FractionalDigits]]` is either **undefined** or a non-negative [integer](#) identifying the number of fractional digits to be used with numeric styles.

13.5 Abstract Operations for DurationFormat Objects

13.5.1 Duration Records

A *Duration Record* is a [Record](#) value used to represent a Duration.

Duration Records have the fields listed in [Table 22](#)

Table 22: [Duration Record](#) Fields

Field	Meaning
<code>[[Years]]</code>	The number of years in the duration.
<code>[[Months]]</code>	The number of months in the duration.
<code>[[Weeks]]</code>	The number of weeks in the duration.
<code>[[Days]]</code>	The number of days in the duration.
<code>[[Hours]]</code>	The number of hours in the duration.
<code>[[Minutes]]</code>	The number of minutes in the duration.
<code>[[Seconds]]</code>	The number of seconds in the duration.
<code>[[Milliseconds]]</code>	The number of milliseconds in the duration.
<code>[[Microseconds]]</code>	The number of microseconds in the duration.
<code>[[Nanoseconds]]</code>	The number of nanoseconds in the duration.

13.5.2 ToIntegerIfIntegral (*argument*)

The abstract operation ToIntegerIfIntegral takes argument *argument* (an ECMAScript language value) and returns either a normal completion containing an integer, or a throw completion. It converts *argument* to an integer representing its Number value, or throws a **RangeError** when that value is not integral. It performs the following steps when called:

1. Let *number* be ? ToNumber(*argument*).
2. If *number* is not an integral Number, throw a **RangeError** exception.
3. Return $\mathbb{R}(\text{number})$.

13.5.3 ToDurationRecord (*input*)

The abstract operation ToDurationRecord takes argument *input* (an ECMAScript language value) and returns either a normal completion containing a Duration Record, or a throw completion. It converts a given object that represents a Duration into a Duration Record. It performs the following steps when called:

1. If *input* is not an Object, then
 - a. If *input* is a String, throw a **RangeError** exception.
 - b. Throw a **TypeError** exception.
2. Let *result* be a new Duration Record with each field set to 0.
3. Let *days* be ? Get(*input*, "days").
4. If *days* is not undefined, set *result*.[[Days]] to ? ToIntegerIfIntegral(*days*).
5. Let *hours* be ? Get(*input*, "hours").
6. If *hours* is not undefined, set *result*.[[Hours]] to ? ToIntegerIfIntegral(*hours*).
7. Let *microseconds* be ? Get(*input*, "microseconds").
8. If *microseconds* is not undefined, set *result*.[[Microseconds]] to ? ToIntegerIfIntegral(*microseconds*).
9. Let *milliseconds* be ? Get(*input*, "milliseconds").
10. If *milliseconds* is not undefined, set *result*.[[Milliseconds]] to ? ToIntegerIfIntegral(*milliseconds*).
11. Let *minutes* be ? Get(*input*, "minutes").
12. If *minutes* is not undefined, set *result*.[[Minutes]] to ? ToIntegerIfIntegral(*minutes*).
13. Let *months* be ? Get(*input*, "months").
14. If *months* is not undefined, set *result*.[[Months]] to ? ToIntegerIfIntegral(*months*).
15. Let *nanoseconds* be ? Get(*input*, "nanoseconds").
16. If *nanoseconds* is not undefined, set *result*.[[Nanoseconds]] to ? ToIntegerIfIntegral(*nanoseconds*).
17. Let *seconds* be ? Get(*input*, "seconds").
18. If *seconds* is not undefined, set *result*.[[Seconds]] to ? ToIntegerIfIntegral(*seconds*).
19. Let *weeks* be ? Get(*input*, "weeks").
20. If *weeks* is not undefined, set *result*.[[Weeks]] to ? ToIntegerIfIntegral(*weeks*).
21. Let *years* be ? Get(*input*, "years").
22. If *years* is not undefined, set *result*.[[Years]] to ? ToIntegerIfIntegral(*years*).
23. If *years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *milliseconds*, *microseconds*, and *nanoseconds* are all undefined, throw a **TypeError** exception.
24. If IsValidDuration(*result*.[[Years]], *result*.[[Months]], *result*.[[Weeks]], *result*.[[Days]], *result*.[[Hours]], *result*.[[Minutes]], *result*.[[Seconds]], *result*.[[Milliseconds]], *result*.[[Microseconds]], *result*.[[Nanoseconds]]) is false, then
 - a. Throw a **RangeError** exception.
25. Return *result*.

13.5.4 DurationSign (*duration*)

The abstract operation DurationSign takes argument *duration* (a Duration Record) and returns -1, 0, or 1. It returns 1 if the most significant non-zero field in the *duration* argument is positive, and -1 if the most significant non-zero field is negative. If all of *duration*'s fields are zero, it returns 0. It performs the following steps when called:

1. For each value *v* of « *duration*.[[Years]], *duration*.[[Months]], *duration*.[[Weeks]], *duration*.[[Days]], *duration*.[[Hours]], *duration*.[[Minutes]], *duration*.[[Seconds]], *duration*.[[Milliseconds]], *duration*.[[Microseconds]], *duration*.[[Nanoseconds]] », do

- a. If $v < 0$, return -1.
 - b. If $v > 0$, return 1.
2. Return 0.

13.5.5 IsValidDuration (*years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *milliseconds*, *microseconds*, *nanoseconds*)

The abstract operation IsValidDuration takes arguments *years* (an integer), *months* (an integer), *weeks* (an integer), *days* (an integer), *hours* (an integer), *minutes* (an integer), *seconds* (an integer), *milliseconds* (an integer), *microseconds* (an integer), and *nanoseconds* (an integer) and returns a Boolean. It returns **true** if its arguments form valid input from which to construct a Duration Record, and **false** otherwise. It performs the following steps when called:

1. Let *sign* be 0.
2. For each value *v* of « *years*, *months*, *weeks*, *days*, *hours*, *minutes*, *seconds*, *milliseconds*, *microseconds*, *nanoseconds* », do
 - a. If $\mathbb{F}(v)$ is not finite, return **false**.
 - b. If $v < 0$, then
 - i. If *sign* > 0 , return **false**.
 - ii. Set *sign* to -1.
 - c. Else if $v > 0$, then
 - i. If *sign* < 0 , return **false**.
 - ii. Set *sign* to 1.
3. If $\text{abs}(\text{years}) \geq 2^{32}$, return **false**.
4. If $\text{abs}(\text{months}) \geq 2^{32}$, return **false**.
5. If $\text{abs}(\text{weeks}) \geq 2^{32}$, return **false**.
6. Let *normalizedSeconds* be $\text{days} \times 86,400 + \text{hours} \times 3600 + \text{minutes} \times 60 + \text{seconds} + \mathbb{R}(\mathbb{F}(\text{milliseconds})) \times 10^{-3} + \mathbb{R}(\mathbb{F}(\text{microseconds})) \times 10^{-6} + \mathbb{R}(\mathbb{F}(\text{nanoseconds})) \times 10^{-9}$.
7. NOTE: The above step cannot be implemented directly using floating-point arithmetic. Multiplying by 10^{-3} , 10^{-6} , and 10^{-9} respectively may be imprecise when *milliseconds*, *microseconds*, or *nanoseconds* is an unsafe integer. This multiplication can be implemented in C++ with an implementation of `std::remquo()` with sufficient bits in the quotient. String manipulation will also give an exact result, since the multiplication is by a power of 10.
8. If $\text{abs}(\text{normalizedSeconds}) \geq 2^{53}$, return **false**.
9. Return **true**.

13.5.6 GetDurationUnitOptions (*unit*, *options*, *baseStyle*, *stylesList*, *digitalBase*, *prevStyle*, *twoDigitHours*)

The abstract operation GetDurationUnitOptions takes arguments *unit* (a String), *options* (an Object), *baseStyle* (a String), *stylesList* (a List of Strings), *digitalBase* (a String), *prevStyle* (a String), and *twoDigitHours* (a Boolean) and returns either a normal completion containing a Duration Unit Options Record, or a throw completion. It extracts the relevant options for any given *unit* from an Object and returns them as a Record. It performs the following steps when called:

1. Let *style* be ? `GetOption(options, unit, STRING, stylesList, undefined)`.
2. Let *displayDefault* be "always".
3. If *style* is undefined, then
 - a. If *baseStyle* is "digital", then
 - i. Set *style* to *digitalBase*.
 - ii. If *unit* is not one of "hours", "minutes", or "seconds", set *displayDefault* to "auto".
 - b. Else if *prevStyle* is one of "fractional", "numeric" or "2-digit", then
 - i. Set *style* to "numeric".
 - ii. If *unit* is not "minutes" or "seconds", set *displayDefault* to "auto".
 - c. Else,
 - i. Set *style* to *baseStyle*.
 - ii. Set *displayDefault* to "auto".

4. If *style* is "numeric" and `IsFractionalSecondUnitName(unit)` is **true**, then
 - a. Set *style* to "fractional".
 - b. Set *displayDefault* to "auto".
5. Let *displayField* be the string-concatenation of *unit* and "Display".
6. Let *display* be ? `GetOption(options, displayField, STRING, « "auto", "always" », displayDefault)`.
7. Perform ? `ValidateDurationUnitStyle(unit, style, display, prevStyle)`.
8. If *unit* is "hours" and *twoDigitHours* is **true**, set *style* to "2-digit".
9. If *unit* is "minutes" or "seconds" and *prevStyle* is "numeric" or "2-digit", set *style* to "2-digit".
10. Return the **Duration Unit Options Record** { `[[Style]]: style`, `[[Display]]: display` }.

13.5.6.1 Duration Unit Options Records

Each *Duration Unit Options Record* has the fields defined in Table 23.

Table 23: Duration Unit Options Record

Field Name	Value Type
<code>[[Style]]</code>	a String from the Styles column of Table 20
<code>[[Display]]</code>	"auto" or "always"

13.5.6.2 ValidateDurationUnitStyle (*unit*, *style*, *display*, *prevStyle*)

The abstract operation `ValidateDurationUnitStyle` takes arguments *unit* (a String), *style* (a String), *display* (a String), and *prevStyle* (a String) and returns either a **normal completion containing** UNUSED or a **throw completion**. It performs the following steps when called:

1. If *display* is "always" and *style* is "fractional", throw a **RangeError** exception.
2. If *prevStyle* is "fractional" and *style* is not "fractional", throw a **RangeError** exception.
3. If *prevStyle* is "numeric" or "2-digit" and *style* is not one of "fractional", "numeric" or "2-digit", throw a **RangeError** exception.
4. Return UNUSED.

NOTE *unit* is not referenced in the preceding algorithm, but it is recommended that implementations use it when constructing the message of a thrown exception.

13.5.7 ComputeFractionalDigits (*durationFormat*, *duration*)

The abstract operation `ComputeFractionalDigits` takes arguments *durationFormat* (a **DurationFormat** Object) and *duration* (a **Duration Record**) and returns a **mathematical value**. It computes the sum of all values in *durationFormat* units with "fractional" style, expressed as a fraction of the smallest unit of *durationFormat* that does not use "fractional" style. It performs the following steps when called:

1. Let *result* be 0.
2. Let *exponent* be 3.
3. For each row of Table 24, except the header row, in table order, do
 - a. Let *unitOptions* be the value of *durationFormat*'s internal slot whose name is the Internal Slot value of the current row.
 - b. If *unitOptions*.`[[Style]]` is "fractional", then
 - i. Let *unit* be the Unit value of the current row.
 - ii. **Assert:** `IsFractionalSecondUnitName(unit)` is **true**.
 - iii. Let *value* be the value of *duration*'s field whose name is the Value Field value of the current row.
 - iv. Set *result* to *result* + (*value* / 10^{*exponent*}).
 - v. Set *exponent* to *exponent* + 3.
4. Return *result*.

13.5.8 NextUnitFractional (*durationFormat*, *unit*)

The abstract operation NextUnitFractional takes arguments *durationFormat* (a DurationFormat Object) and *unit* (a String) and returns a Boolean. It returns **true** if the next smallest unit uses the "fractional" style. It performs the following steps when called:

1. If *unit* is "seconds" and *durationFormat*.[[MillisecondsOptions]].[[Style]] is "fractional", return **true**.
2. If *unit* is "milliseconds" and *durationFormat*.[[MicrosecondsOptions]].[[Style]] is "fractional", return **true**.
3. If *unit* is "microseconds" and *durationFormat*.[[NanosecondsOptions]].[[Style]] is "fractional", return **true**.
4. Return **false**.

13.5.9 FormatNumericHours (*durationFormat*, *hoursValue*, *signDisplayed*)

The abstract operation FormatNumericHours takes arguments *durationFormat* (a DurationFormat object), *hoursValue* (an integer), and *signDisplayed* (a Boolean) and returns a List of Records. *hoursValue* is an integer indicating a number of hours. It creates the parts for *hoursValue* according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. Let *result* be a new empty List.
2. Let *hoursStyle* be *durationFormat*.[[HoursOptions]].[[Style]].
3. Assert: *hoursStyle* is "numeric" or *hoursStyle* is "2-digit".
4. Let *nfOpts* be OrdinaryObjectCreate(null).
5. Let *numberingSystem* be *durationFormat*.[[NumberingSystem]].
6. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "numberingSystem", *numberingSystem*).
7. If *hoursStyle* is "2-digit", then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "minimumIntegerDigits", 2_F).
8. If *signDisplayed* is false, then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "signDisplay", "never").
9. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "useGrouping", false).
10. Let *nf* be ! Construct(%Intl.NumberFormat%, « *durationFormat*.[[Locale]], *nfOpts* »).
11. Let *hoursParts* be PartitionNumberPattern(*nf*, *hoursValue*).
12. For each Record { [[Type]], [[Value]] } *part* of *hoursParts*, do
 - a. Append the Record { [[Type]]: *part*.[[Type]], [[Value]]: *part*.[[Value]], [[Unit]]: "hour" } to *result*.
13. Return *result*.

13.5.10 FormatNumericMinutes (*durationFormat*, *minutesValue*, *hoursDisplayed*, *signDisplayed*)

The abstract operation FormatNumericMinutes takes arguments *durationFormat* (a DurationFormat Object), *minutesValue* (an integer), *hoursDisplayed* (a Boolean), and *signDisplayed* (a Boolean) and returns a List of Records. *minutesValue* is an integer indicating a number of minutes. It creates the parts for *minutesValue* according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. Let *result* be a new empty List.
2. If *hoursDisplayed* is true, then
 - a. Let *separator* be *durationFormat*.[[HourMinuteSeparator]].
 - b. Append the Record { [[Type]]: "literal", [[Value]]: *separator*, [[Unit]]: EMPTY } to *result*.
3. Let *minutesStyle* be *durationFormat*.[[MinutesOptions]].[[Style]].
4. Assert: *minutesStyle* is "numeric" or *minutesStyle* is "2-digit".
5. Let *nfOpts* be OrdinaryObjectCreate(null).
6. Let *numberingSystem* be *durationFormat*.[[NumberingSystem]].
7. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "numberingSystem", *numberingSystem*).
8. If *minutesStyle* is "2-digit", then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "minimumIntegerDigits", 2_F).
9. If *signDisplayed* is false, then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "signDisplay", "never").
10. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "useGrouping", false).
11. Let *nf* be ! Construct(%Intl.NumberFormat%, « *durationFormat*.[[Locale]], *nfOpts* »).
12. Let *minutesParts* be PartitionNumberPattern(*nf*, *minutesValue*).

13. For each **Record** { [[Type]], [[Value]] } *part* of *minutesParts*, do
 - a. Append the **Record** { [[Type]]: *part*.[[Type]], [[Value]]: *part*.[[Value]], [[Unit]]: "minute" } to *result*.
14. Return *result*.

13.5.11 FormatNumericSeconds (*durationFormat*, *secondsValue*, *minutesDisplayed*, *signDisplayed*)

The abstract operation FormatNumericSeconds takes arguments *durationFormat* (a DurationFormat Object), *secondsValue* (a mathematical value), *minutesDisplayed* (a Boolean), and *signDisplayed* (a Boolean) and returns a List of Records. *secondsValue* is a mathematical value indicating a number of seconds. It creates the parts for *secondsValue* according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. Let *result* be a new empty List.
2. If *minutesDisplayed* is **true**, then
 - a. Let *separator* be *durationFormat*.[[MinuteSecondSeparator]].
 - b. Append the **Record** { [[Type]]: "literal", [[Value]]: *separator*, [[Unit]]: EMPTY } to *result*.
3. Let *secondsStyle* be *durationFormat*.[[SecondsOptions]].[[Style]].
4. **Assert**: *secondsStyle* is "numeric" or *secondsStyle* is "2-digit".
5. Let *nfOpts* be OrdinaryObjectCreate(null).
6. Let *numberingSystem* be *durationFormat*.[[NumberingSystem]].
7. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "numberingSystem", *numberingSystem*).
8. If *secondsStyle* is "2-digit", then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "minimumIntegerDigits", 2_F).
9. If *signDisplayed* is **false**, then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "signDisplay", "never").
10. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "useGrouping", false).
11. Let *fractionDigits* be *durationFormat*.[[FractionalDigits]].
12. If *fractionDigits* is **undefined**, then
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "maximumFractionDigits", 9_F).
 - b. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "minimumFractionDigits", +0_F).
13. Else,
 - a. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "maximumFractionDigits", *fractionDigits*).
 - b. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "minimumFractionDigits", *fractionDigits*).
14. Perform ! CreateDataPropertyOrThrow(*nfOpts*, "roundingMode", "trunc").
15. Let *nf* be ! Construct(%Intl.NumberFormat%, « *durationFormat*.[[Locale]], *nfOpts* »).
16. Let *secondsParts* be PartitionNumberPattern(*nf*, *secondsValue*).
17. For each **Record** { [[Type]], [[Value]] } *part* of *secondsParts*, do
 - a. Append the **Record** { [[Type]]: *part*.[[Type]], [[Value]]: *part*.[[Value]], [[Unit]]: "second" } to *result*.
18. Return *result*.

13.5.12 FormatNumericUnits (*durationFormat*, *duration*, *firstNumericUnit*, *signDisplayed*)

The abstract operation FormatNumericUnits takes arguments *durationFormat* (a DurationFormat Object), *duration* (a Duration Record), *firstNumericUnit* (a String), and *signDisplayed* (a Boolean) and returns a List of Records. It creates the parts representing the elements of *duration* that use "numeric" or "2-digit" style according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. **Assert**: *firstNumericUnit* is "hours", "minutes", or "seconds".
2. Let *numericPartsList* be a new empty List.
3. Let *hoursValue* be *duration*.[[Hours]].
4. Let *hoursDisplay* be *durationFormat*.[[HoursOptions]].[[Display]].
5. Let *minutesValue* be *duration*.[[Minutes]].
6. Let *minutesDisplay* be *durationFormat*.[[MinutesOptions]].[[Display]].
7. Let *secondsValue* be *duration*.[[Seconds]].
8. If *duration*.[[Milliseconds]] is not 0 or *duration*.[[Microseconds]] is not 0 or *duration*.[[Nanoseconds]] is not 0, then
 - a. Set *secondsValue* to *secondsValue* + ComputeFractionalDigits(*durationFormat*, *duration*).
9. Let *secondsDisplay* be *durationFormat*.[[SecondsOptions]].[[Display]].
10. Let *hoursFormatted* be **false**.

11. If *firstNumericUnit* is **"hours"**, then
 - a. If *hoursValue* is not 0 or *hoursDisplay* is **"always"**, then
 - i. Set *hoursFormatted* to **true**.
12. If *secondsValue* is not 0 or *secondsDisplay* is **"always"**, then
 - a. Let *secondsFormatted* be **true**.
13. Else,
 - a. Let *secondsFormatted* be **false**.
14. Let *minutesFormatted* be **false**.
15. If *firstNumericUnit* is **"hours"** or *firstNumericUnit* is **"minutes"**, then
 - a. If *hoursFormatted* is **true** and *secondsFormatted* is **true**, then
 - i. Set *minutesFormatted* to **true**.
 - b. Else if *minutesValue* is not 0 or *minutesDisplay* is **"always"**, then
 - i. Set *minutesFormatted* to **true**.
16. If *hoursFormatted* is **true**, then
 - a. If *signDisplayed* is **true**, then
 - i. If *hoursValue* is 0 and *DurationSign*(*duration*) is -1, then
 1. Set *hoursValue* to NEGATIVE-ZERO.
 - b. Let *hoursParts* be *FormatNumericHours*(*durationFormat*, *hoursValue*, *signDisplayed*).
 - c. Set *numericPartsList* to the list-concatenation of *numericPartsList* and *hoursParts*.
 - d. Set *signDisplayed* to **false**.
17. If *minutesFormatted* is **true**, then
 - a. If *signDisplayed* is **true**, then
 - i. If *minutesValue* is 0 and *DurationSign*(*duration*) is -1, then
 1. Set *minutesValue* to NEGATIVE-ZERO.
 - b. Let *minutesParts* be *FormatNumericMinutes*(*durationFormat*, *minutesValue*, *hoursFormatted*, *signDisplayed*).
 - c. Set *numericPartsList* to the list-concatenation of *numericPartsList* and *minutesParts*.
 - d. Set *signDisplayed* to **false**.
18. If *secondsFormatted* is **true**, then
 - a. Let *secondsParts* be *FormatNumericSeconds*(*durationFormat*, *secondsValue*, *minutesFormatted*, *signDisplayed*).
 - b. Set *numericPartsList* to the list-concatenation of *numericPartsList* and *secondsParts*.
19. Return *numericPartsList*.

13.5.13 IsFractionalSecondUnitName (*unit*)

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

1. If *unit* is one of **"milliseconds"**, **"microseconds"**, or **"nanoseconds"**, return **true**.
2. Return **false**.

13.5.14 ListFormatParts (*durationFormat*, *partitionedPartsList*)

The abstract operation *ListFormatParts* takes arguments *durationFormat* (a *DurationFormat* Object) and *partitionedPartsList* (a List of Lists of Records) and returns a List. It creates a List corresponding to the parts within the Lists in *partitionedPartsList* according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. Let *IfOpts* be *OrdinaryObjectCreate*(**null**).
2. Perform ! *CreateDataPropertyOrThrow*(*IfOpts*, **"type"**, **"unit"**).
3. Let *listStyle* be *durationFormat*.[[Style]].
4. If *listStyle* is **"digital"**, then
 - a. Set *listStyle* to **"short"**.
5. Perform ! *CreateDataPropertyOrThrow*(*IfOpts*, **"style"**, *listStyle*).
6. Let *If* be ! *Construct*(%Intl.ListFormat%, « *durationFormat*.[[Locale]], *IfOpts* »).
7. Let *strings* be a new empty List.
8. For each element *parts* of *partitionedPartsList*, do
 - a. Let *string* be the empty String.

- b. For each **Record** { [[Type]], [[Value]], [[Unit]] } *part* in *parts*, do
 - i. Set *string* to the string-concatenation of *string* and *part*.[[Value]].
 - c. Append *string* to *strings*.
 9. Let *formattedPartsList* be **CreatePartsFromList**(*If*, *strings*).
 10. Let *partitionedPartsIndex* be 0.
 11. Let *partitionedLength* be the number of elements in *partitionedPartsList*.
 12. Let *flattenedPartsList* be a new empty **List**.
 13. For each **Record** { [[Type]], [[Value]], [[Unit]] } *listPart* in *formattedPartsList*, do
 - a. If *listPart*.[[Type]] is **"element"**, then
 - i. **Assert**: *partitionedPartsIndex* < *partitionedLength*.
 - ii. Let *parts* be *partitionedPartsList*[*partitionedPartsIndex*].
 - iii. For each **Record** { [[Type]], [[Value]], [[Unit]] } *part* in *parts*, do
 1. Append *part* to *flattenedPartsList*.
 - iv. Set *partitionedPartsIndex* to *partitionedPartsIndex* + 1.
 - b. Else,
 - i. **Assert**: *listPart*.[[Type]] is **"literal"**.
 - ii. Append the **Record** { [[Type]]: **"literal"**, [[Value]]: *listPart*.[[Value]], [[Unit]]: **EMPTY** } to *flattenedPartsList*.
 14. Return *flattenedPartsList*.

13.5.15 PartitionDurationFormatPattern (*durationFormat*, *duration*)

The abstract operation **PartitionDurationFormatPattern** takes arguments *durationFormat* (a **DurationFormat**) and *duration* (a **Duration Record**) and returns a **List**. It creates the corresponding parts for *duration* according to the effective locale and the formatting options of *durationFormat*. It performs the following steps when called:

1. Let *result* be a new empty **List**.
2. Let *signDisplayed* be **true**.
3. Let *numericUnitFound* be **false**.
4. While *numericUnitFound* is **false**, repeat for each row in **Table 24** in table order, except the header row:
 - a. Let *value* be the value of *duration*'s field whose name is the Value Field value of the current row.
 - b. Let *unitOptions* be the value of *durationFormat*'s internal slot whose name is the Internal Slot value of the current row.
 - c. Let *style* be *unitOptions*.[[Style]].
 - d. Let *display* be *unitOptions*.[[Display]].
 - e. Let *unit* be the Unit value of the current row.
 - f. Let *numberFormatUnit* be the NumberFormat Unit value of the current row.
 - g. If *style* is **"numeric"** or **"2-digit"**, then
 - i. Let *numericPartsList* be **FormatNumericUnits**(*durationFormat*, *duration*, *unit*, *signDisplayed*).
 - ii. If *numericPartsList* is not empty, append *numericPartsList* to *result*.
 - iii. Set *numericUnitFound* to **true**.
 - h. Else,
 - i. Let *nfOpts* be **OrdinaryObjectCreate**(**null**).
 - ii. If **NextUnitFractional**(*durationFormat*, *unit*) is **true**, then
 1. Set *value* to *value* + **ComputeFractionalDigits**(*durationFormat*, *duration*).
 2. Let *fractionDigits* be *durationFormat*.[[FractionalDigits]].
 3. If *fractionDigits* is **undefined**, then
 - a. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"maximumFractionDigits"**, **9_F**).
 - b. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"minimumFractionDigits"**, **+0_F**).
 4. Else,
 - a. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"maximumFractionDigits"**, *fractionDigits*).
 - b. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"minimumFractionDigits"**, *fractionDigits*).
 5. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"roundingMode"**, **"trunc"**).
 6. Set *numericUnitFound* to **true**.
 - iii. If *display* is **"always"** or *value* is not 0, then
 1. Perform ! **CreateDataPropertyOrThrow**(*nfOpts*, **"numberingSystem"**, *durationFormat*.[[NumberingSystem]]).
 2. If *signDisplayed* is **true**, then

- a. Set *signDisplayed* to **false**.
 - b. If *value* is 0 and *DurationSign*(*duration*) is -1, set *value* to NEGATIVE-ZERO.
 3. Else,
 - a. Perform ! *CreateDataPropertyOrThrow*(*nfOpts*, "signDisplay", "never").
 4. Perform ! *CreateDataPropertyOrThrow*(*nfOpts*, "style", "unit").
 5. Perform ! *CreateDataPropertyOrThrow*(*nfOpts*, "unit", *numberFormatUnit*).
 6. Perform ! *CreateDataPropertyOrThrow*(*nfOpts*, "unitDisplay", *style*).
 7. Let *nf* be ! *Construct*(%Intl.NumberFormat%, « *durationFormat*.[[Locale]], *nfOpts* »).
 8. Let *parts* be *PartitionNumberPattern*(*nf*, *value*).
 9. Let *list* be a new empty List.
 10. For each *Record* { [[Type]], [[Value]] } *part* of *parts*, do
 - a. Append the *Record* { [[Type]]: *part*.[[Type]], [[Value]]: *part*.[[Value]], [[Unit]]: *numberFormatUnit* } to *list*.
 11. Append *list* to *result*.
5. Return *ListFormatParts*(*durationFormat*, *result*).

Table 24: DurationFormat instance internal slots and properties relevant to PartitionDurationFormatPattern

Value Field	Internal Slot	Unit	NumberFormat Unit
[[Years]]	[[YearsOptions]]	"years"	"year"
[[Months]]	[[MonthsOptions]]	"months"	"month"
[[Weeks]]	[[WeeksOptions]]	"weeks"	"week"
[[Days]]	[[DaysOptions]]	"days"	"day"
[[Hours]]	[[HoursOptions]]	"hours"	"hour"
[[Minutes]]	[[MinutesOptions]]	"minutes"	"minute"
[[Seconds]]	[[SecondsOptions]]	"seconds"	"second"
[[Milliseconds]]	[[MillisecondsOptions]]	"milliseconds"	"millisecond"
[[Microseconds]]	[[MicrosecondsOptions]]	"microseconds"	"microsecond"
[[Nanoseconds]]	[[NanosecondsOptions]]	"nanoseconds"	"nanosecond"

14 ListFormat Objects

14.1 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.

14.1.1 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*.

14.2 Properties of the Intl.ListFormat Constructor

The Intl.ListFormat [constructor](#) has the following properties:

14.2.1 Intl.ListFormat.prototype

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

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

14.2.2 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*).

14.2.3 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").

14.3 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.

14.3.1 Intl.ListFormat.prototype.constructor

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

14.3.2 Intl.ListFormat.prototype.resolvedOptions ()

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

1. Let *if* be the **this** value.
2. Perform ? [RequireInternalSlot](#)(*if*, [[InitializedListFormat]]).
3. Let *options* be [OrdinaryObjectCreate](#)(%Object.prototype%).
4. For each row of [Table 25](#), 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 *if*'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 25: Resolved Options of ListFormat Instances

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

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

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

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

14.3.4 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*).

14.3.5 Intl.ListFormat.prototype [%Symbol.toStringTag%]

The initial value of the [%Symbol.toStringTag%](#) property is the String value **"Intl.ListFormat"**.

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

14.4 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 Intl.ListFormat 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](#).

14.5 Abstract Operations for ListFormat Objects

14.5.1 DeconstructPattern (*pattern*, *placeables*)

The abstract operation DeconstructPattern takes arguments *pattern* (a [Pattern 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 *subst* is a *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*.

14.5.2 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 the *Record* { [[Type]]: "element", [[Value]]: *list*[0] }.
 - d. Let *second* be the *Record* { [[Type]]: "element", [[Value]]: *list*[1] }.
 - e. Let *placeables* be the *Record* { [[0]]: *first*, [[1]]: *second* }.
 - f. Return *DeconstructPattern(pattern, placeables)*.
4. Let *last* be the *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 the *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 the *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.

14.5.3 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 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*.

14.5.4 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*.

14.5.5 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 *next* is not a 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.

15 Locale Objects

15.1 The Intl.Locale Constructor

The Locale [constructor](#) is the %Intl.Locale% intrinsic object and a standard built-in property of the Intl object.

15.1.1 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 *localeExtensionKeys* be **%Intl.Locale%[[LocaleExtensionKeys]]**.
3. Let *internalSlotsList* be « **[[InitializedLocale]]**, **[[Locale]]**, **[[Calendar]]**, **[[Collation]]**, **[[HourCycle]]**, **[[NumberingSystem]]** ».
4. If *localeExtensionKeys* contains **"kf"**, then
 - a. Append **[[CaseFirst]]** to *internalSlotsList*.
5. If *localeExtensionKeys* contains **"kn"**, then
 - a. Append **[[Numeric]]** to *internalSlotsList*.
6. Let *locale* be ? **OrdinaryCreateFromConstructor**(**NewTarget**, **"%Intl.Locale.prototype%"**, *internalSlotsList*).
7. If *tag* is not a **String** and *tag* is not an **Object**, throw a **TypeError** exception.
8. If *tag* is an **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. If **IsStructurallyValidLanguageTag**(*tag*) is **false**, throw a **RangeError** exception.
12. Set *tag* to **CanonicalizeUnicodeLocaleId**(*tag*).
13. Set *tag* to ? **UpdateLanguageId**(*tag*, *options*).
14. Let *opt* be a new **Record**.
15. Let *calendar* be ? **GetOption**(*options*, **"calendar"**, **STRING**, **EMPTY**, **undefined**).
16. If *calendar* is not **undefined**, then
 - a. If *calendar* cannot be matched by the **type Unicode locale nonterminal**, throw a **RangeError** exception.
17. Set *opt*.**[[ca]]** to *calendar*.
18. Let *collation* be ? **GetOption**(*options*, **"collation"**, **STRING**, **EMPTY**, **undefined**).
19. If *collation* is not **undefined**, then
 - a. If *collation* cannot be matched by the **type Unicode locale nonterminal**, throw a **RangeError** exception.
20. Set *opt*.**[[co]]** to *collation*.
21. Let *hc* be ? **GetOption**(*options*, **"hourCycle"**, **STRING**, « **"h11"**, **"h12"**, **"h23"**, **"h24"** », **undefined**).
22. Set *opt*.**[[hc]]** to *hc*.
23. Let *kf* be ? **GetOption**(*options*, **"caseFirst"**, **STRING**, « **"upper"**, **"lower"**, **"false"** », **undefined**).
24. Set *opt*.**[[kf]]** to *kf*.
25. Let *kn* be ? **GetOption**(*options*, **"numeric"**, **BOOLEAN**, **EMPTY**, **undefined**).
26. If *kn* is not **undefined**, set *kn* to ! **ToString**(*kn*).
27. Set *opt*.**[[kn]]** to *kn*.
28. Let *numberingSystem* be ? **GetOption**(*options*, **"numberingSystem"**, **STRING**, **EMPTY**, **undefined**).
29. If *numberingSystem* is not **undefined**, then
 - a. If *numberingSystem* cannot be matched by the **type Unicode locale nonterminal**, throw a **RangeError** exception.
30. Set *opt*.**[[nu]]** to *numberingSystem*.
31. Let *r* be **MakeLocaleRecord**(*tag*, *opt*, *localeExtensionKeys*).
32. Set *locale*.**[[Locale]]** to *r*.**[[locale]]**.
33. Set *locale*.**[[Calendar]]** to *r*.**[[ca]]**.
34. Set *locale*.**[[Collation]]** to *r*.**[[co]]**.
35. Set *locale*.**[[HourCycle]]** to *r*.**[[hc]]**.
36. If *localeExtensionKeys* contains **"kf"**, then
 - a. Set *locale*.**[[CaseFirst]]** to *r*.**[[kf]]**.
37. If *localeExtensionKeys* 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**.
38. Set *locale*.**[[NumberingSystem]]** to *r*.**[[nu]]**.
39. Return *locale*.

15.1.2 UpdateLanguageId (*tag*, *options*)

The abstract operation UpdateLanguageId takes arguments *tag* (a Unicode canonicalized locale identifier) and *options* (an Object) and returns either a normal completion containing a language tag or a throw completion. It updates the **unicode_language_id** subtags in *tag* from the corresponding properties of *options* and returns the structurally valid but non-canonicalized result. It performs the following steps when called:

1. Let *baseName* be **GetLocaleBaseName**(*tag*).
2. Let *language* be ? **GetOption**(*options*, "language", STRING, EMPTY, **GetLocaleLanguage**(*baseName*)).
3. 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, **GetLocaleScript**(*baseName*)).
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, **GetLocaleRegion**(*baseName*)).
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. Let *variants* be **GetLocaleVariants**(*baseName*).
9. Let *allExtensions* be the suffix of *tag* following *baseName*.
10. Let *newTag* be *language*.
11. If *script* is not **undefined**, set *newTag* to the string-concatenation of *newTag*, "-", and *script*.
12. If *region* is not **undefined**, set *newTag* to the string-concatenation of *newTag*, "-", and *region*.
13. If *variants* is not **undefined**, set *newTag* to the string-concatenation of *newTag*, "-", and *variants*.
14. Set *newTag* to the string-concatenation of *newTag* and *allExtensions*.
15. Return *newTag*.

15.1.3 MakeLocaleRecord (*tag*, *options*, *localeExtensionKeys*)

The abstract operation MakeLocaleRecord takes arguments *tag* (a language tag), *options* (a Record), and *localeExtensionKeys* (a List of Strings) and returns a Record. It constructs and returns a Record in which each element of *localeExtensionKeys* defines a corresponding field with data from any Unicode locale extension sequence of *tag* as overridden by a corresponding field of *options*, and which additionally includes a **[[locale]]** field containing a Unicode canonicalized locale identifier resulting from incorporating those fields into *tag*. It performs the following steps when called:

1. 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]]**.
2. Else,
 - a. Let *attributes* be a new empty List.
 - b. Let *keywords* be a new empty List.
3. Let *result* be a new Record.
4. For each element *key* of *localeExtensionKeys*, do
 - a. If *keywords* contains an element whose **[[Key]]** is *key*, then
 - i. Let *entry* be the element of *keywords* whose **[[Key]]** is *key*.
 - ii. Let *value* be *entry*.**[[Value]]**.
 - b. Else,
 - i. Let *entry* be **EMPTY**.
 - ii. Let *value* be **undefined**.
 - c. **Assert**: *options* has a field **[[<key>]]**.
 - d. Let *overrideValue* be *options*.**[[<key>]]**.
 - e. If *overrideValue* is not **undefined**, then
 - i. Set *value* to **CanonicalizeUValue**(*key*, *overrideValue*).

- ii. If *entry* is not EMPTY, then
 1. Set *entry*.[[Value]] to *value*.
- iii. Else,
 1. Append the **Record** { [[Key]]: *key*, [[Value]]: *value* } to *keywords*.
- f. Set *result*.[[<key>]] to *value*.
5. Let *locale* be the String value that is *tag* with any **Unicode locale extension sequences** removed.
6. If *attributes* is not empty or *keywords* is not empty, then
 - a. Set *result*.[[locale]] to **InsertUnicodeExtensionAndCanonicalize**(*locale*, *attributes*, *keywords*).
7. Else,
 - a. Set *result*.[[locale]] to **CanonicalizeUnicodeLocaleId**(*locale*).
8. Return *result*.

15.2 Properties of the Intl.Locale Constructor

The Intl.Locale **constructor** has the following properties:

15.2.1 Intl.Locale.prototype

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

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

15.2.2 Internal slots

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

15.3 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.

15.3.1 Intl.Locale.prototype.constructor

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

15.3.2 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. Return **GetLocaleBaseName**(*loc*.[[Locale]]).

15.3.3 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]].

15.3.4 get Intl.Locale.prototype.caseFirst

This property only exists if `%Intl.Locale%.[[LocaleExtensionKeys]]` 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]].

15.3.5 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]].

15.3.6 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]].

15.3.7 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]]).

15.3.8 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*).

15.3.9 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*).

15.3.10 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]].

15.3.11 get Intl.Locale.prototype.numeric

This property only exists if [%Intl.Locale%](#).[[LocaleExtensionKeys]] 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]].

15.3.12 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]]).

15.3.13 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]]).

15.3.14 Intl.Locale.prototype.toString ()

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

15.3.15 Intl.Locale.prototype [%Symbol.toStringTag%]

The initial value of the [%Symbol.toStringTag%](#) property is the String value **"Intl.Locale"**.

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

15.4 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 Intl.Locale Constructor](#):

- `[[Locale]]` is a [String](#) value with the [language tag](#) of the locale whose localization is used for formatting.
- `[[Calendar]]` is either **undefined** or a String value that is a well-formed [Unicode Calendar Identifier](#) `<https://unicode.org/reports/tr35/#UnicodeCalendarIdentifier>` in canonical form.
- `[[Collation]]` is either **undefined** or a String value that is a well-formed [Unicode Collation Identifier](#) `<https://unicode.org/reports/tr35/#UnicodeCollationIdentifier>` in canonical form.
- `[[HourCycle]]` is either **undefined** or a String value that is a valid [Unicode Hour Cycle Identifier](#) `<https://unicode.org/reports/tr35/#UnicodeHourCycleIdentifier>` in canonical form.
- `[[NumberingSystem]]` is either **undefined** or a String value that is a well-formed [Unicode Number System Identifier](#) `<https://unicode.org/reports/tr35/#UnicodeNumberSystemIdentifier>` in canonical form.
- `[[CaseFirst]]` is either **undefined** or one of the String values **"upper"**, **"lower"**, or **"false"**. This internal slot only exists if the `[[LocaleExtensionKeys]]` internal slot of [%Intl.Locale%](#) contains **"kf"**.
- `[[Numeric]]` is either **undefined** or a Boolean value specifying whether numeric sorting is used by the locale. This internal slot only exists if the `[[LocaleExtensionKeys]]` internal slot of [%Intl.Locale%](#) contains **"kn"**.

15.5 Abstract Operations for Locale Objects

15.5.1 GetLocaleBaseName (*locale*)

The abstract operation `GetLocaleBaseName` 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. Return the longest prefix of *locale* matched by the **unicode_language_id** [Unicode locale nonterminal](#).

15.5.2 GetLocaleLanguage (*locale*)

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

1. Let *baseName* be `GetLocaleBaseName(locale)`.
2. **Assert:** The first [subtag](#) of *baseName* can be matched by the **unicode_language_subtag** [Unicode locale nonterminal](#).
3. Return the first [subtag](#) of *baseName*.

15.5.3 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. Let *baseName* be `GetLocaleBaseName(locale)`.
2. **Assert:** *baseName* contains at most one [subtag](#) that can be matched by the **unicode_script_subtag** [Unicode locale nonterminal](#).
3. If *baseName* contains a [subtag](#) matched by the **unicode_script_subtag** [Unicode locale nonterminal](#), return that [subtag](#).
4. Return **undefined**.

15.5.4 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. Let *baseName* be `GetLocaleBaseName(locale)`.
2. **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.
3. **Assert:** The first [subtag](#) of *baseName* can be matched by the **unicode_language_subtag** [Unicode locale nonterminal](#).

4. Let *baseNameTail* be the suffix of *baseName* following the first *subtag*.
5. **Assert:** *baseNameTail* contains at most one *subtag* that can be matched by the **unicode_region_subtag** *Unicode locale nonterminal*.
6. If *baseNameTail* contains a *subtag* matched by the **unicode_region_subtag** *Unicode locale nonterminal*, return that *subtag*.
7. Return **undefined**.

15.5.5 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. Let *baseName* be GetLocaleBaseName(*locale*).
2. **NOTE:** Each *subtag* in *baseName* that is preceded by "-" is either a **unicode_script_subtag**, **unicode_region_subtag**, or **unicode_variant_subtag**, but any substring matched by **unicode_variant_subtag** is strictly longer than any prefix thereof which could also be matched by one of the other productions.
3. Let *variants* be the longest suffix of *baseName* that starts with a "-" followed by a substring that is matched by the **unicode_variant_subtag** *Unicode locale nonterminal*. If there is no such suffix, return **undefined**.
4. Return the *substring* of *variants* from 1.

16 NumberFormat Objects

16.1 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.

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

15. Perform ? *SetNumberFormatUnitOptions*(*numberFormat*, *options*).
16. Let *style* be *numberFormat*.[[Style]].
17. Let *notation* be ? *GetOption*(*options*, "notation", STRING, « "standard", "scientific", "engineering", "compact" », "standard").
18. Set *numberFormat*.[[Notation]] to *notation*.
19. If *style* is "currency" and *notation* is "standard", then
 - a. Let *currency* be *numberFormat*.[[Currency]].
 - b. Let *cDigits* be *CurrencyDigits*(*currency*).
 - c. Let *mnfdDefault* be *cDigits*.
 - d. Let *mxfdDefault* be *cDigits*.
20. Else,
 - a. Let *mnfdDefault* be 0.
 - b. If *style* is "percent", then
 - i. Let *mxfdDefault* be 0.
 - c. Else,
 - i. Let *mxfdDefault* be 3.
21. Perform ? *SetNumberFormatDigitOptions*(*numberFormat*, *options*, *mnfdDefault*, *mxfdDefault*, *notation*).
22. Let *compactDisplay* be ? *GetOption*(*options*, "compactDisplay", STRING, « "short", "long" », "short").
23. Let *defaultUseGrouping* be "auto".
24. If *notation* is "compact", then
 - a. Set *numberFormat*.[[CompactDisplay]] to *compactDisplay*.
 - b. Set *defaultUseGrouping* to "min2".
25. NOTE: For historical reasons, the strings "true" and "false" are accepted and replaced with the default value.
26. Let *useGrouping* be ? *GetBooleanOrStringNumberFormatOption*(*options*, "useGrouping", « "min2", "auto", "always", "true", "false" », *defaultUseGrouping*).
27. If *useGrouping* is "true" or *useGrouping* is "false", set *useGrouping* to *defaultUseGrouping*.
28. If *useGrouping* is true, set *useGrouping* to "always".
29. Set *numberFormat*.[[UseGrouping]] to *useGrouping*.
30. Let *signDisplay* be ? *GetOption*(*options*, "signDisplay", STRING, « "auto", "never", "always", "exceptZero", "negative" », "auto").
31. Set *numberFormat*.[[SignDisplay]] to *signDisplay*.
32. 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*).
33. Return *numberFormat*.

NORMATIVE OPTIONAL

16.1.1.1 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*.

16.1.2 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 16.5.3). 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 *intlObj*.[[MinimumFractionDigits]], throw a **RangeError** exception.
30. Return UNUSED.

16.1.3 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.

16.2 Properties of the Intl.NumberFormat Constructor

The Intl.NumberFormat [constructor](#) has the following properties:

16.2.1 Intl.NumberFormat.prototype

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

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

16.2.2 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*)**.

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 « **"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) <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 es2025, 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/>).

16.3 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.

16.3.1 Intl.NumberFormat.prototype.constructor

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

16.3.2 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 26](#), 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 $\mathbb{F}(v)$.
 - ii. Perform ! [CreateDataPropertyOrThrow](#)(*options*, *p*, *v*).
6. Return *options*.

Table 26: Resolved Options of NumberFormat Instances

Internal Slot	Property	Conversion
<code>[[Locale]]</code>	<code>"locale"</code>	
<code>[[NumberingSystem]]</code>	<code>"numberingSystem"</code>	
<code>[[Style]]</code>	<code>"style"</code>	
<code>[[Currency]]</code>	<code>"currency"</code>	
<code>[[CurrencyDisplay]]</code>	<code>"currencyDisplay"</code>	
<code>[[CurrencySign]]</code>	<code>"currencySign"</code>	
<code>[[Unit]]</code>	<code>"unit"</code>	

Table 26: Resolved Options of NumberFormat Instances (*continued*)

Internal Slot	Property	Conversion
[[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"	

16.3.3 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 (16.5.2).
 - 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.

16.3.4 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*).

16.3.5 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*).

16.3.6 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*).

16.3.7 Intl.NumberFormat.prototype [%Symbol.toStringTag%]

The initial value of the [%Symbol.toStringTag%](#) property is the String value **"Intl.NumberFormat"**.

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

16.4 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 Intl.NumberFormat 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 representing the [Unicode Number System Identifier](#) <<https://unicode.org/reports/tr35/#UnicodeNumberSystemIdentifier>> 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 [16.3.2](#) 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 [ILD](#) 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 a *rounding mode*, one of the String values in the Identifier column of [Table 27](#).
- `[[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 27: 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]

Table 27: Rounding modes in Intl.NumberFormat (continued)

Identifier	Description	Examples: Round to 0 fraction digits				
		-1.5	0.4	0.5	0.6	1.5
"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 (16.3.3).

16.5 Abstract Operations for NumberFormat Objects

16.5.1 CurrencyDigits (*currency*)

The [implementation-defined](#) 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. Return a non-negative integer indicating the number of fractional digits used when formatting quantities of the currency corresponding to *currency*. If there is no available information on the number of digits to be used, return 2.

16.5.2 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:** *nf* is an 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_F.

16.5.3 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 $fResult.[[RoundingMagnitude]] < sResult.[[RoundingMagnitude]]$, let $fixedIsMorePrecise$ be **true**; else let $fixedIsMorePrecise$ be **false**.
 - d. If $intlObject.[[RoundingType]]$ is MORE-PRECISION and $fixedIsMorePrecise$ is **true**, then
 - i. Let $result$ be $fResult$.
 - e. Else if $intlObject.[[RoundingType]]$ is LESS-PRECISION and $fixedIsMorePrecise$ is **false**, then
 - i. Let $result$ be $fResult$.
 - f. Else,
 - i. Let $result$ be $sResult$.
8. Set x to $result.[[RoundedNumber]]$.
9. Let $string$ be $result.[[FormattedString]]$.
10. If $intlObject.[[TrailingZeroDisplay]]$ is "stripIfInteger" and $x \bmod 1 = 0$, then
 - a. Let i be `StringIndexOf(string, ".", 0)`.
 - b. If i is not NOT-FOUND, 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 , $[[FormattedString]]$: $string$ }.

16.5.4 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 `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 $100 \times x$.
 - iii. Set $exponent$ to `ComputeExponent( $numberFormat$ ,  $x$ )`.
 - iv. Set x to $x \times 10^{-exponent}$.
 - 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 "number", then
 - i. Let *notationSubParts* be *PartitionNotationSubPattern*(*numberFormat*, *x*, *n*, *exponent*).
 - ii. Set *result* to the list-concatenation of *result* and *notationSubParts*.
 - d. Else if *p* is "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 "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 "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 "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 "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 "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 "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 "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*.

16.5.5 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 "number", then
 1. If the *numberFormat*.[[NumberingSystem]] matches one of the values in the Numbering System column of Table 28 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 28.
 - 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 $0x0030 \leq c \leq 0x0039$, 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* is not NOT-FOUND and *decimalSepIndex* > 0, then
 - a. Let *integer* be the substring of *n* from 0 to *decimalSepIndex*.
 - b. Let *fraction* be the substring of *n* from *decimalSepIndex* + 1.
 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 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 **"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 **"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 **"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 **"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 28: 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
gara	U+10D40 to U+10D49
gong	U+11DA0 to U+11DA9
gonm	U+11D50 to U+11D59
gujr	U+0AE6 to U+0AEF

Table 28: Numbering systems with simple digit mappings (continued)

Numbering System	Digits
gukh	U+16130 to U+16139
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
krai	U+16D70 to U+16D79
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
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
mymrepka	U+116DA to U+116E3
mymrpao	U+116D0 to U+116D9
mymrshan	U+1090 to U+1099
mymrtlng	U+A9F0 to U+A9F9

Table 28: Numbering systems with simple digit mappings (continued)

Numbering System	Digits
nagm	U+1E4F0 to U+1E4F9
newa	U+11450 to U+11459
nkoo	U+07C0 to U+07C9
olck	U+1C50 to U+1C59
onao	U+1E5F1 to U+1E5FA
orya	U+0B66 to U+0B6F
osma	U+104A0 to U+104A9
outlined	U+1CCF0 to U+1CCF9
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
sunu	U+11BF0 to U+11BF9
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
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 [ILD](#) and [ILND](#) String values and locations within numeric strings that depend on the effective locale of [numberFormat](#), or upon the effective locale and numbering system of [numberFormat](#). The [ILD](#) and [ILND](#) String values 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/>).

16.5.6 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*.

16.5.7 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(*F*(*n*)), *O*).
 - e. Increment *n* by 1.
5. Return *result*.

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

The abstract operation ToRawPrecision takes arguments *x* (a 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 29, 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:

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

where $10^{p-1} \leq n < 10^p$

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 *xFinal* be ApplyUnsignedRoundingMode(*x*, *r1*, *r2*, *unsignedRoundingMode*).
 - d. If *xFinal* is *r1*, then

- i. Let n be $n1$.
 - ii. Let e be $e1$.
- e. Else,
 - i. Let n be $n2$.
 - ii. Let e be $e2$.
- f. Let m be the String consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
4. If $e \geq (p - 1)$, 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 $e \geq 0$, 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]]: $e - p + 1$ }.

16.5.9 ToRawFixed (x , $minFraction$, $maxFraction$, $roundingIncrement$, $unsignedRoundingMode$)

The abstract operation ToRawFixed takes arguments x (a 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 29, 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 $r1 = \text{ToRawFixedFn}(n1, f)$, such that $n1 \bmod \text{roundingIncrement} = 0$, $r1 \leq x$, and $r1$ is maximized.
3. Let $n2$ be an integer and $r2$ a mathematical value, with $r2 = \text{ToRawFixedFn}(n2, f)$, such that $n2 \bmod \text{roundingIncrement} = 0$, $r2 \geq x$, and $r2$ is minimized.
4. Let $xFinal$ be $\text{ApplyUnsignedRoundingMode}(x, r1, r2, unsignedRoundingMode)$.
5. If $xFinal$ is $r1$, let n be $n1$. Otherwise, let n be $n2$.
6. 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).
7. If $f \neq 0$, then
 - a. Let k be the length of m .
 - b. If $k \leq f$, 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*.
8. Else,
 - a. Let *int* be the length of *m*.
9. Let *cut* be *maxFraction* - *minFraction*.
10. 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.
11. If the last code unit of *m* is 0x002E (FULL STOP), then
 - a. Remove the last code unit from *m*.
12. Return the Record { [[FormattedString]]: *m*, [[RoundedNumber]]: *xFinal*, [[IntegerDigitsCount]]: *int*, [[RoundingMagnitude]]: -*f* }.

NORMATIVE OPTIONAL

16.5.10 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 *nf* is not an 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*.

16.5.11 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 16.2.3. 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 16.2.3).
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*.

16.5.12 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 16.2.3. 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 16.2.3).
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}".

16.5.13 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 $x \times 10^{-\text{exponent}}$.
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).

16.5.14 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 ILD integer by which to scale a number of the given magnitude in compact notation for the current locale.
 - c. Return *exponent*.

16.5.15 Runtime Semantics: StringIntIMV

The [syntax-directed operation](#) StringIntIMV 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 [StringIntIMV](#) of *StrNumericLiteral*.

StrNumericLiteral ::: *NonDecimalIntegerLiteral*

1. Return MV of *NonDecimalIntegerLiteral*.

StrDecimalLiteral ::: - *StrUnsignedDecimalLiteral*

1. Let *a* be [StringIntIMV](#) 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 $a \times 10^e$.

16.5.16 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 *primValue* is a BigInt, return $\mathbb{R}(\text{primValue})$.
3. If *primValue* is a String, then
 - a. Let *str* be *primValue*.
4. Else,
 - a. Let *x* be ? ToNumber(*primValue*).
 - b. If *x* is $-\mathbf{0}_{\mathbb{F}}$, 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 $+\infty_{\mathbb{F}}$ and *intlMV* < 0, return NEGATIVE-INFINITY.
 - c. If *rounded* is $+\infty_{\mathbb{F}}$, return POSITIVE-INFINITY.
 - d. If *rounded* is $+\mathbf{0}_{\mathbb{F}}$ and *intlMV* < 0, return NEGATIVE-ZERO.
 - e. If *rounded* is $+\mathbf{0}_{\mathbb{F}}$, return 0.
10. Return *intlMV*.

16.5.17 GetUnsignedRoundingMode (*roundingMode*, *sign*)

The abstract operation GetUnsignedRoundingMode takes arguments *roundingMode* (a rounding mode) and *sign* (NEGATIVE or POSITIVE) and returns a specification type from the Unsigned Rounding Mode column of Table 29. It returns the rounding mode that should be applied to the absolute value of a number to produce the same result as if *roundingMode* 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 29 for the row where the value in the Identifier column is *roundingMode* and the value in the Sign column is *sign*.

Table 29: 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

Table 29: Conversion from **rounding mode** to unsigned **rounding mode**
(continued)

Identifier	Sign	Unsigned Rounding Mode
"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

16.5.18 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 29, 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 *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 *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 $(r1 / (r2 - r1)) \bmod 2$.
15. If *cardinality* is 0, return *r1*.
16. Return *r2*.

16.5.19 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 *xResult* be PartitionNumberPattern(*numberFormat*, *x*).
3. Let *yResult* be PartitionNumberPattern(*numberFormat*, *y*).

4. If `FormatNumeric(numberFormat, x)` is `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`.
5. Let `result` be a new empty `List`.
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(numberFormat, result)`.

16.5.20 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 the `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`.

16.5.21 CollapseNumberRange (numberFormat, result)

The `implementation-defined` abstract operation `CollapseNumberRange` takes arguments `numberFormat` (an `Intl.NumberFormat`) and `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`) according to the effective locale and the formatting options of `numberFormat` by removing redundant information, resolving internal inconsistency, replacing characters when necessary, and inserting spacing when necessary. It then returns the resulting `List`. The algorithm is `ILND`, 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`.

16.5.22 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*.

16.5.23 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($\mathbb{F}(n)$), *O*).
 - f. Increment *n* by 1.
5. Return *result*.

17 PluralRules Objects

17.1 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.

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

17.2 Properties of the Intl.PluralRules Constructor

The Intl.PluralRules constructor has the following properties:

17.2.1 Intl.PluralRules.prototype

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

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

17.2.2 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*).

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 « ».

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) <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/>).

17.3 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.

17.3.1 Intl.PluralRules.prototype.constructor

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

17.3.2 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]], sorted according to the following order: "zero", "one", "two", "few", "many", "other".
5. For each row of Table 30, 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 $\mathbb{F}(v)$.
 - ii. Perform ! **CreateDataPropertyOrThrow**($options$, p , v).
6. Return $options$.

Table 30: 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"	

17.3.3 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]].

17.3.4 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).

17.3.5 Intl.PluralRules.prototype [%Symbol.toStringTag%]

The initial value of the `%Symbol.toStringTag%` property is the String value **"Intl.PluralRules"**.

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

17.4 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 Intl.PluralRules 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 [16.4](#).
- `[[ComputedRoundingPriority]]` is one of the String values **"auto"**, **"morePrecision"**, or **"lessPrecision"**. It is only used in [17.3.2](#) 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]]` identifies the **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).

17.5 Abstract Operations for PluralRules Objects

17.5.1 PluralRuleSelect (*locale*, *type*, *s*)

The **implementation-defined** abstract operation `PluralRuleSelect` takes arguments *locale* (a **language tag**), *type* (**"cardinal"** or **"ordinal"**), and *s* (a decimal String) and returns **"zero"**, **"one"**, **"two"**, **"few"**, **"many"**, or **"other"**. The returned String characterizes the plural category of *s* according to the effective locale and the options of *pluralRules*.

17.5.2 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 *res* be `FormatNumericToString(pluralRules,  $\mathbb{R}(n)$ )`.
3. Let *s* be *res*.`[[FormattedString]]`.
4. Let *locale* be *pluralRules*.`[[Locale]]`.

5. Let *type* be *pluralRules*.[[Type]].
6. Let *p* be *PluralRuleSelect*(*locale*, *type*, *s*).
7. Return the *Record* { [[PluralCategory]]: *p*, [[FormattedString]]: *s* }.

17.5.3 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".

17.5.4 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]]).

18 RelativeTimeFormat Objects

18.1 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.

18.1.1 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. Let *requestedLocales* be ? *CanonicalizeLocaleList*(*locales*).
4. Set *options* to ? *CoerceOptionsToObject*(*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 *numberingSystem* be ? *GetOption*(*options*, "numberingSystem", STRING, EMPTY, **undefined**).
9. If *numberingSystem* is not **undefined**, then
 - a. If *numberingSystem* cannot be matched by the **type** *Unicode locale nonterminal*, throw a **RangeError** exception.
10. Set *opt*.[[nu]] to *numberingSystem*.

11. Let *r* be `ResolveLocale(%Intl.RelativeTimeFormat%.[[AvailableLocales]], requestedLocales, opt, %Intl.RelativeTimeFormat%.[[RelevantExtensionKeys]], %Intl.RelativeTimeFormat%.[[LocaleData]])`.
12. Let *locale* be *r*.`[[Locale]]`.
13. Set *relativeTimeFormat*.`[[Locale]]` to *locale*.
14. Set *relativeTimeFormat*.`[[LocaleData]]` to *r*.`[[LocaleData]]`.
15. Set *relativeTimeFormat*.`[[NumberingSystem]]` to *r*.`[[nu]]`.
16. Let *style* be ? `GetOption(options, "style", STRING, « "long", "short", "narrow" », "long")`.
17. Set *relativeTimeFormat*.`[[Style]]` to *style*.
18. Let *numeric* be ? `GetOption(options, "numeric", STRING, « "always", "auto" », "always")`.
19. Set *relativeTimeFormat*.`[[Numeric]]` to *numeric*.
20. Let *nfOptions* be `OrdinaryObjectCreate(null)`.
21. Perform ! `CreateDataPropertyOrThrow(nfOptions, "numberingSystem", relativeTimeFormat.[[NumberingSystem]])`.
22. Let *relativeTimeFormat*.`[[NumberFormat]]` be ! `Construct(%Intl.NumberFormat%, « locale, nfOptions »)`.
23. Let *relativeTimeFormat*.`[[PluralRules]]` be ! `Construct(%Intl.PluralRules%, « locale »)`.
24. Return *relativeTimeFormat*.

18.2 Properties of the Intl.RelativeTimeFormat Constructor

The Intl.RelativeTimeFormat [constructor](#) has the following properties:

18.2.1 Intl.RelativeTimeFormat.prototype

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

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

18.2.2 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)`.

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 « **"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) <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/>).

18.3 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.

18.3.1 Intl.RelativeTimeFormat.prototype.constructor

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

18.3.2 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 31](#), 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 31: Resolved Options of RelativeTimeFormat Instances

Internal Slot	Property
<code>[[Locale]]</code>	"locale"
<code>[[Style]]</code>	"style"
<code>[[Numeric]]</code>	"numeric"
<code>[[NumberingSystem]]</code>	"numberingSystem"

18.3.3 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*).

18.3.4 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*).

18.3.5 Intl.RelativeTimeFormat.prototype [%Symbol.toStringTag%]

The initial value of the [%Symbol.toStringTag%](#) property is the String value **"Intl.RelativeTimeFormat"**.

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

18.4 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 Intl.RelativeTimeFormat 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 representing the [Unicode Number System Identifier](#) <<https://unicode.org/reports/tr35/#UnicodeNumberSystemIdentifier>> used for formatting.
- [[PluralRules]] is an Intl.PluralRules object used for formatting.

18.5 Abstract Operations for RelativeTimeFormat Objects

18.5.1 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*.

18.5.2 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, $+\infty$, or $-\infty$, throw a **RangeError** exception.
2. Let *unit* be ? SingularRelativeTimeUnit(*unit*).
3. Let *fields* be *relativeTimeFormat*.[[LocaleData]].
4. Let *patterns* be *fields*.[[<*unit*>]].
5. Let *style* be *relativeTimeFormat*.[[Style]].
6. If *style* is "short" or "narrow", then
 - a. Let *key* be the string-concatenation of *unit*, "-", and *style*.
 - b. If *fields* has a field [[<*key*>]], set *patterns* to *fields*.[[<*key*>]].
7. If *relativeTimeFormat*.[[Numeric]] is "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*, *[[Unit]]*: EMPTY }.
8. If *value* is $-\infty$ or *value* < $-\infty$, then
 - a. Let *tl* be "past".
9. Else,
 - a. Let *tl* be "future".
10. Let *po* be *patterns*.[[<*tl*>]].
11. Let *fv* be PartitionNumberPattern(*relativeTimeFormat*.[[NumberFormat]], $\mathbb{R}$ (*value*)).
12. Let *pr* be ResolvePlural(*relativeTimeFormat*.[[PluralRules]], *value*).[[PluralCategory]].
13. Let *pattern* be *po*.[[<*pr*>]].
14. Return MakePartsList(*pattern*, *unit*, *fv*).

18.5.3 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 }
»
```

18.5.4 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 the 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*.

18.5.5 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 EMPT, then
 - i. Perform ! [CreateDataPropertyOrThrow](#)(*O*, "unit", *part*.[[Unit]]).
 - e. Perform ! [CreateDataPropertyOrThrow](#)(*result*, ! [ToString](#)(*n*), *O*).
 - f. Increment *n* by 1.
5. Return *result*.

19 Segmenter Objects

19.1 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](#).

19.1.1 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*.

19.2 Properties of the Intl.Segmenter Constructor

The Intl.Segmenter *constructor* has the following properties:

19.2.1 Intl.Segmenter.prototype

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

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

19.2.2 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*).

19.2.3 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.

19.3 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.

19.3.1 Intl.Segmenter.prototype.constructor

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

19.3.2 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 32, 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 32: Resolved Options of Segmenter Instances

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

19.3.3 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*).

19.3.4 Intl.Segmenter.prototype [%Symbol.toStringTag%]

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

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

19.4 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 Intl.Segmenter 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.

19.5 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.

19.5.1 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*.

19.5.2 The %IntlSegmentsPrototype% Object

The *%IntlSegmentsPrototype%* object:

- is the prototype of all Segments objects.
- is an [ordinary object](#).
- has the following properties:

19.5.2.1 %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*).

19.5.2.2 %IntlSegmentsPrototype% [%Symbol.iterator%] ()

The **%Symbol.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*).

19.5.3 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.

19.6 Segment Iterator Objects

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

19.6.1 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*.

19.6.2 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:

19.6.2.1 %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 *len* be the length of *string*.
7. If *startIndex* ≥ *len*, then
 - a. Return [CreateIteratorResultObject](#)(**undefined**, **true**).
8. Let *endIndex* be [FindBoundary](#)(*segmenter*, *string*, *startIndex*, AFTER).
9. Set *iterator*.[[IteratedStringNextSegmentCodeUnitIndex]] to *endIndex*.
10. Let *segmentData* be [CreateSegmentDataObject](#)(*segmenter*, *string*, *startIndex*, *endIndex*).
11. Return [CreateIteratorResultObject](#)(*segmentData*, **false**).

19.6.2.2 %IntlSegmentIteratorPrototype% [%Symbol.toStringTag%]

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

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

19.6.3 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 33](#).

Table 33: 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.

19.7 Segment Data Objects

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

19.7.1 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: *endIndex* ≤ *len*.
3. Assert: *startIndex* < *endIndex*.
4. Let *result* be OrdinaryObjectCreate(%Object.prototype%).
5. Let *segment* be the substring of *string* from *startIndex* to *endIndex*.
6. Perform ! CreateDataPropertyOrThrow(*result*, "segment", *segment*).
7. Perform ! CreateDataPropertyOrThrow(*result*, "index", $\mathbb{F}(\text{startIndex})$).
8. Perform ! CreateDataPropertyOrThrow(*result*, "input", *string*).
9. Let *granularity* be *segmenter*.[[SegmenterGranularity]].
10. 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*).
11. 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".

19.8 Abstract Operations for Segmenter Objects

19.8.1 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. 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. It performs the following steps when called:

1. Let *len* be the length of *string*.
2. Assert: *startIndex* < *len*.
3. Let *locale* be *segmenter*.[[Locale]].
4. Let *granularity* be *segmenter*.[[SegmenterGranularity]].
5. If *direction* is BEFORE, then
 - a. 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*.
 - b. If a boundary is found, return the count of code units in *string* preceding it.
 - c. Return 0.
6. Assert: *direction* is AFTER.
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>>).

20 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.

20.1 Properties of the String Prototype Object

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

This definition supersedes the definition provided in es2025, 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*).

The "**`length`**" property of this function is 1_F.

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.

20.1.2 `String.prototype.toLocaleLowerCase` ([*locales*])

This definition supersedes the definition provided in es2025, 22.1.3.26.

This function interprets a String value as a sequence of code points, as described in es2025, 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.

20.1.2.1 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 es2025, 6.1.4, and returns the result of 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 *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.
5. Let *match* be LookupMatchingLocaleByPrefix(*availableLocales*, « *requestedLocale* »).
6. If *match* is not undefined, let *locale* be *match*.[[locale]]; else let *locale* be "und".
7. Let *codePoints* be StringToCodePoints(*S*).
8. 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.
9. 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.
10. 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) <<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()**.

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

This definition supersedes the definition provided in es2025, 22.1.3.27.

This function interprets a String value as a sequence of code points, as described in es2025, 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.

20.2 Properties of the Number Prototype Object

The following definition(s) refer to the abstract operation `thisNumberValue` as defined in es2025, 21.1.3.

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

This definition supersedes the definition provided in es2025, 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))`.

20.3 Properties of the BigInt Prototype Object

The following definition(s) refer to the abstract operation `thisBigIntValue` as defined in es2025, 21.2.3.

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

This definition supersedes the definition provided in es2025, 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,  $\mathbb{R}(x)$ )`.

20.4 Properties of the Date Prototype Object

The following definition(s) refer to the abstract operation `thisTimeValue` as defined in es2025, 21.4.4.

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

This definition supersedes the definition provided in es2025, 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)`.

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

This definition supersedes the definition provided in es2025, 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*).

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

This definition supersedes the definition provided in es2025, 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*).

20.5 Properties of the Array Prototype Object

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

This definition supersedes the definition provided in es2025, 23.1.3.32.

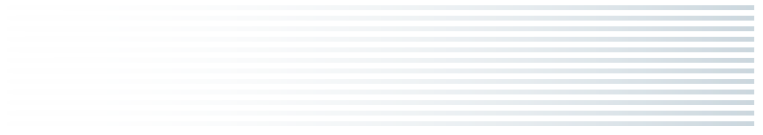
When the **toLocaleString** method is called with optional arguments *locales* and *options*, the following steps are taken:

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*(*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 es2025, 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 (9.2.4)
- In Collator:
 - Support for the Unicode extensions keys **"kf"**, **"kn"** and the parallel options properties **"caseFirst"**, **"numeric"** (10.1.1)
 - 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 (11.5.5)
 - The calendric calculations used for calendars other than **"gregory"** (11.5.12)
 - 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 (21.4.1.19)
- In DisplayNames:
 - The localized names (12.2.3)
- In DurationFormat:
 - The set of supported **"nu"** key values (numbering systems) per locale (13.2.3)
 - The digital formatting configuration (use of two-digit hours with style **"numeric"** and separators for numeric hours, minutes, and seconds) per locale (13.2.3)
- In ListFormat:
 - The patterns used for formatting values (14.2.3)
- In Locale:
 - Support for the Unicode extensions keys **"kf"**, **"kn"** and the parallel options properties **"caseFirst"**, **"numeric"** (15.1.1)
- In NumberFormat:
 - The set of supported **"nu"** key values (numbering systems) per locale (16.2.3)
 - 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 (16.5.6)
 - The number of fractional digits used when formatting currency values (16.5.6)
 - Localized representations of **NaN** and **Infinity** (16.5.6)
 - The implementation of numbering systems not listed in Table 28 (16.5.6)
 - Localized decimal and grouping separators (16.5.6)
 - Localized plus and minus signs (16.5.6)
 - Localized digit grouping schemata (16.5.6)
 - Localized magnitude thresholds for compact notation (16.5.6)
 - Localized symbols for compact and scientific notation (16.5.6)
 - Localized narrow, short, and long currency symbols and names (16.5.6)

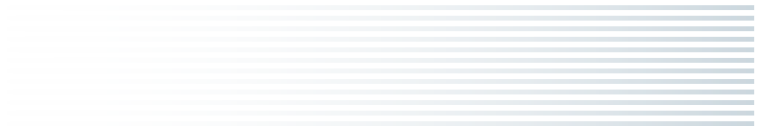
- Localized narrow, short, and long unit symbols ([16.5.6](#))
- In PluralRules:
 - [List](#) of Strings representing the possible results of plural selection and their corresponding order per locale. ([17.1.1](#))
- In RelativeTimeFormat:
 - The set of supported **"nu"** key values (numbering systems) per locale ([18.2.3](#))
 - The patterns used for formatting values ([18.2.3](#))
- In Segmenter:
 - Boundary determination algorithms ([19.8.1](#))
 - Classification of segments as "word-like" ([19.7.1](#))

Annex B

(informative)

Additions and Changes That Introduce Incompatibilities with Prior Editions

- [10.1](#), [16.1](#), [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_F**. In 2nd Edition, **"length"** is set to **1_F**.
- [10.3.4](#) In ECMA-402, 7th Edition, the [%Symbol.toStringTag%](#) property of **Intl.Collator.prototype** was set to **"Object"**. In 8th Edition, [%Symbol.toStringTag%](#) is set to **"Intl.Collator"**.
- [11.3.7](#) In ECMA-402, 7th Edition, the [%Symbol.toStringTag%](#) property of **Intl.DateTimeFormat.prototype** was set to **"Object"**. In 8th Edition, [%Symbol.toStringTag%](#) is set to **"Intl.DateTimeFormat"**.
- [16.3.7](#) In ECMA-402, 7th Edition, the [%Symbol.toStringTag%](#) property of **Intl.NumberFormat.prototype** was set to **"Object"**. In 8th Edition, [%Symbol.toStringTag%](#) is set to **"Intl.NumberFormat"**.
- [17.3.5](#) In ECMA-402, 7th Edition, the [%Symbol.toStringTag%](#) property of **Intl.PluralRules.prototype** was set to **"Object"**. In 8th Edition, [%Symbol.toStringTag%](#) is set to **"Intl.PluralRules"**.
- [8.1.1](#) In ECMA-402, 7th Edition, the [%Symbol.toStringTag%](#) property of **Intl** was not defined. In 8th Edition, [%Symbol.toStringTag%](#) is set to **"Intl"**.
- [16.1](#) 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.



Annex C

(informative)

Colophon

This specification is authored on [GitHub](https://github.com/tc39/ecma402) <<https://github.com/tc39/ecma402>> in a plaintext source format called [Ecmarkup](https://github.com/bterlson/ecmarkup) <<https://github.com/bterlson/ecmarkup>>. Ecmarkup is an HTML and Markdown dialect that provides a framework and toolset for authoring Ecma 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) <<https://github.com/rbuckton/grammarkdown>> for defining syntax and [Ecmarkdown](https://github.com/domenic/ecmarkdown) <<https://github.com/domenic/ecmarkdown>> for authoring algorithm steps. PDF renderings of this specification are produced using [Prince for Books](https://www.prince-xml.com/books/) <<https://www.prince-xml.com/books/>>, the book publishing-optimized version of [PrinceXML](https://www.princexml.com/) <<https://www.princexml.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.

This Ecma Standard was developed by Technical Committee 39 and was adopted by the General Assembly of June 2025.



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/>

All Software contained in this document ("Software") is protected by copyright and is being made available under the "BSD License", included below. This Software may be subject to third party rights (rights from parties other than Ecma International), including patent rights, and no licenses under such third party rights are granted under this license even if the third party concerned is a member of Ecma International. SEE THE ECMA CODE OF CONDUCT IN PATENT MATTERS AVAILABLE AT <https://ecma-international.org/memento/codeofconduct.htm> FOR INFORMATION REGARDING THE LICENSING OF PATENT CLAIMS THAT ARE REQUIRED TO IMPLEMENT ECMA INTERNATIONAL STANDARDS.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the authors nor Ecma International may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE ECMA INTERNATIONAL "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL ECMA INTERNATIONAL BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.