ECMAScript® 2021 Internationalization API Specification
 Contributing to this Specification

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

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

TC39-TG2:
Convener: Shane F. Carr (@sffc)
Admin group: contact by email

Editors:
Leo Balter (@leobalter) Richard Gibson (@gibson042)

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scope</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Conformance</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Normative references</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Overview</td>
<td>2</td>
</tr>
<tr>
<td>4.1</td>
<td>Internationalization, Localization, and Globalization</td>
<td>2</td>
</tr>
<tr>
<td>4.2</td>
<td>API Overview</td>
<td>3</td>
</tr>
<tr>
<td>4.3</td>
<td>API Conventions</td>
<td>3</td>
</tr>
<tr>
<td>4.4</td>
<td>Implementation Dependencies</td>
<td>3</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Compatibility across implementations</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Notational Conventions</td>
<td>4</td>
</tr>
<tr>
<td>5.1</td>
<td>Well-Known Intrinsic Objects</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>Identification of Locales, Currencies, TimeZones, and Measurement Units</td>
<td>5</td>
</tr>
<tr>
<td>6.1</td>
<td>Case Sensitivity and Case Mapping</td>
<td>5</td>
</tr>
<tr>
<td>6.2</td>
<td>Language Tags</td>
<td>5</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Unicode Locale Extension Sequences</td>
<td>6</td>
</tr>
<tr>
<td>6.2.2</td>
<td>IsStructurallyValidLanguageTag (locale)</td>
<td>6</td>
</tr>
<tr>
<td>6.2.3</td>
<td>CanonicalizeUnicodeLocaleId (locale)</td>
<td>6</td>
</tr>
<tr>
<td>6.2.4</td>
<td>DefaultLocale ()</td>
<td>7</td>
</tr>
<tr>
<td>6.3</td>
<td>Currency Codes</td>
<td>7</td>
</tr>
<tr>
<td>6.3.1</td>
<td>IsValidCurrencyCode (currency)</td>
<td>7</td>
</tr>
<tr>
<td>6.4</td>
<td>Time Zone Names</td>
<td>8</td>
</tr>
<tr>
<td>6.4.1</td>
<td>IsValidTimeZoneName (timeZone)</td>
<td>8</td>
</tr>
<tr>
<td>6.4.2</td>
<td>CanonicalizeTimeZoneName</td>
<td>8</td>
</tr>
<tr>
<td>6.4.3</td>
<td>DefaultTimeZone ()</td>
<td>8</td>
</tr>
<tr>
<td>6.5</td>
<td>Measurement Unit Identifiers</td>
<td>8</td>
</tr>
<tr>
<td>6.5.1</td>
<td>IsWellFormedUnitIdentifier (unitIdentifier)</td>
<td>9</td>
</tr>
<tr>
<td>6.5.2</td>
<td>IsSanctionedSimpleUnitIdentifier (unitIdentifier)</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>Requirements for Standard Built-inECMAScript Objects</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>The Intl Object</td>
<td>11</td>
</tr>
<tr>
<td>8.1</td>
<td>Value Properties of the Intl Object</td>
<td>11</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Intl[@@toStringTag]</td>
<td>11</td>
</tr>
<tr>
<td>8.2</td>
<td>Constructor Properties of the Intl Object</td>
<td>11</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Intl.Collator ( . . )</td>
<td>11</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Intl.DateTimeFormat ( . . )</td>
<td>11</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Intl.DisplayNames ( . . )</td>
<td>11</td>
</tr>
<tr>
<td>8.2.4</td>
<td>Intl.ListFormat ( . . )</td>
<td>12</td>
</tr>
<tr>
<td>8.2.5</td>
<td>Intl.Locale ( . . )</td>
<td>12</td>
</tr>
<tr>
<td>8.2.6</td>
<td>Intl.NumberFormat ( . . )</td>
<td>12</td>
</tr>
<tr>
<td>8.2.7</td>
<td>Intl.PluralRules ( . . )</td>
<td>12</td>
</tr>
<tr>
<td>8.2.8</td>
<td>Intl.RelativeTimeFormat ( . . )</td>
<td>12</td>
</tr>
<tr>
<td>8.3</td>
<td>Function Properties of the Intl Object</td>
<td>12</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Intl.getCanonicalLocales (locales)</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>Locale and Parameter Negotiation</td>
<td>12</td>
</tr>
<tr>
<td>9.1</td>
<td>Internal slots of Service Constructors</td>
<td>12</td>
</tr>
<tr>
<td>9.2</td>
<td>Abstract Operations</td>
<td>13</td>
</tr>
<tr>
<td>9.2.1</td>
<td>CanonicalizeLocaleList (locales)</td>
<td>13</td>
</tr>
<tr>
<td>9.2.2</td>
<td>BestAvailableLocale (availableLocales, locale)</td>
<td>14</td>
</tr>
<tr>
<td>9.2.3</td>
<td>LookupMatcher (availableLocales, requestedLocales)</td>
<td>14</td>
</tr>
</tbody>
</table>
Locale Sensitive Functions of the ECMAScript Language Specification ........................................86
18.1 Properties of the String Prototype Object ........................................................................86
18.1.1 String.prototype.localeCompare ( that [ , locales [ , options ] ] ) ........................................86
18.1.2 String.prototype.toLocaleLowerCase ( [ locales ] ) ..........................................................86
18.1.3 String.prototype.toLocaleUpperCase ( [ locales ] ) .........................................................87
18.2 Properties of the Number Prototype Object ....................................................................87
18.2.1 Number.prototype.toLocaleString ( [ locales [ , options ] ] ) ..............................................87
18.3 Properties of the Bigint Prototype Object ........................................................................88
18.3.1 BigInt.prototype.toLocaleString ( [ locales [ , options ] ] ) ..................................................88
18.4 Properties of the Date Prototype Object ............................................................................88
18.4.1 Date.prototype.toLocaleString ( [ locales [ , options ] ] ) .....................................................88
18.4.2 Date.prototype.toLocaleDateString ( [ locales [ , options ] ] ) ............................................88
18.4.3 Date.prototype.toLocaleTimeString ( [ locales [ , options ] ] ) .............................................88
18.5 Properties of the Array Prototype Object .........................................................................89
18.5.1 Array.prototype.toLocaleString ( [ locales [ , options ] ] ) ..................................................89
Annex A Implementation Dependent Behaviour ........................................................................91
Annex B Additions and Changes That Introduce Incompatibilities with Prior Editions ..........93
Annex C Colophon ..................................................................................................................95
Annex D Copyright & Software License ..................................................................................97
Introduction

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

The ECMAScript 2021 Internationalization API Specification (ECMA-402 8th Edition), provides key language sensitive functionality as a complement to the ECMAScript 2021 Language Specification (ECMA-262 12th Edition or successor). Its functionality has been selected from that of well-established internationalization APIs such as those of the Internationalization Components for Unicode (ICU) library, of the .NET framework, or of the Java platform.

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

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

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

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

Unfortunately, it is impossible to identify and acknowledge every person and organization who has contributed to this effort.

Norbert Lindenberg
ECMA-402, 1st Edition Project Editor

Rick Waldron
ECMA-402, 2nd Edition Project Editor

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

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

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

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

This Ecma Standard was developed by Technical Committee 39 and was adopted by the General Assembly of June 2021.
"COPYRIGHT NOTICE

© 2021 Ecma International

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."

Software License

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://www.ecma-international.org/memento/codeofconduct.htm FOR INFORMATION REGARDING THE LICENSING OF PATENT CLAIMS THAT ARE REQUIRED TO IMPLEMENTECMA 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.
ECMAScript® 2021 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 the ECMAScript 2021 Internationalization API Specification must conform to the ECMAScript 2021 Language Specification (ECMA-262 12th Edition, or successor), and must provide and support all the objects, properties, functions, and program semantics described in this specification.

A conforming implementation of the ECMAScript 2021 Internationalization API Specification is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of the ECMAScript 2021 Internationalization API Specification is permitted to provide properties not described in this specification, and values for those properties, for objects that are described in this specification. A conforming implementation is not permitted to add optional arguments to the functions defined in this specification.

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

- The `options` property "localeMatcher" in all constructors and `supportedLocalesOf` methods.
- The `options` properties "usage" and "sensitivity" in the Collator constructor.
- The `options` properties "style", "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 4 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.


NOTE Throughout this document, the phrase “ES2021, x” (where x is a sequence of numbers separated by periods) may be used as shorthand for “ECMAScript 2021 Language Specification (ECMA-262 12th Edition, sub clause x)”.

ISO/IEC 10646:2014: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2015 and Amendment 2, plus additional amendments and corrigenda, or successor

- https://www.iso.org/iso/catalogue_detail.htm?csnumber=63182
- https://www.iso.org/iso/catalogue_detail.htm?csnumber=65047
- https://www.iso.org/iso/catalogue_detail.htm?csnumber=66791

ISO 4217:2015, Codes for the representation of currencies and funds, or successor

IETF RFC 4647, Matching of Language Tags, or successor

IANA Time Zone Database

The Unicode Standard

Unicode Technical Standard 35

- Unicode Locale Data Markup Language
- Unicode BCP 47 Locale Identifiers

4 Overview

This section contains a non-normative overview of the ECMAScript 2021 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 enabling worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization of software is commonly understood to be the combination of internationalization and localization. Globalization starts at the lowest level by using a text representation that supports all languages in the world, and using standard identifiers to identify languages, countries, time zones, and other relevant parameters. It continues with using a user interface language and data presentation that the user understands, and finally often requires product-specific adaptations to the user’s language, culture, and environment.

The ECMAScript 2021 Language Specification lays the foundation by using Unicode for text representation and by providing a few language-sensitive functions, but gives applications little control over the behaviour of these functions. The ECMAScript 2021 Internationalization API Specification builds on this by providing a set of customizable language-sensitive functionality. The API is useful even for applications that themselves are not internationalized, as even applications targeting only one language and one region need to properly support that one language and region. However, the API also enables applications that support multiple languages and regions, even concurrently, as may be needed in server environments.
4.2 API Overview

The ECMAScript 2021 Internationalization API Specification is designed to complement the ECMAScript 2021 Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript 2021 Language Specification (ECMA-262 12th Edition, or successor).

The ECMAScript 2021 Internationalization API Specification provides several key pieces of language-sensitive functionality that are required in most applications: String comparison (collation), number formatting, date and time formatting, display names, list formatting, pluralization rules, and case conversion. While the ECMAScript 2021 Language Specification provides functions for this basic functionality (on Array.prototype: toLocaleString; on String.prototype: localeCompare, toLocaleLowerCase, toLocaleUpperCase; on Number.prototype: toLocaleString; on Date.prototype: toLocaleString, toLocaleDateString, and toLocaleTimeString), it leaves the actual behaviour of these functions largely up to implementations to define. The ECMAScript 2021 Internationalization API Specification provides additional functionality, control over the language and over details of the behaviour to be used, and a more complete specification of required functionality.

Applications can use the API in two ways:

1. Directly, by using the constructors Intl.Collator, Intl.DateTimeFormat, Intl.DisplayNames, Intl.ListFormat, Intl.NumberFormat, Intl.PluralRules, or Intl.RelativeTimeFormat to construct an object, specifying a list of preferred languages and options to configure the behaviour of the resulting object. The object then provides a main function (compare, select, or format), which can be called repeatedly. It also provides a resolvedOptions function, which the application can use to find out the exact configuration of the object.

2. Indirectly, by using the functions of the ECMAScript 2021 Language Specification mentioned above. The collation and formatting functions are respecified in this specification to accept the same arguments as the Collator, NumberFormat, and DateTimeFormat constructors and produce the same results as their compare or format methods. The case conversion functions are respecified to accept a list of preferred languages.

The Intl object is used to package all functionality defined in the ECMAScript 2021 Internationalization API Specification 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.

4.3 API Conventions

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

4.4 Implementation Dependencies

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

- The set of locales that an implementation supports with adequate localizations: Linguists estimate the number of human languages to around 6000, and the more widely spoken ones have variations based on regions or other parameters. Even large locale data collections, such as the Common Locale Data Repository, cover only a subset of this large set. Implementations targeting resource-constrained devices may have to further reduce the subset.

- The exact form of localizations such as format patterns: In many cases locale-dependent conventions are not standardized, so different forms may exist side by side, or they vary over time. Different
internationalization libraries may have implemented different forms, without any of them being actually wrong. In order to allow this API to be implemented on top of existing libraries, such variations have to be permitted.

- **Subsets of Unicode:** Some operations, such as collation, operate on strings that can include characters from the entire Unicode character set. However, both the Unicode standard and the ECMAScript standard allow implementations to limit their functionality to subsets of the Unicode character set. In addition, locale conventions typically don't specify the desired behaviour for the entire Unicode character set, but only for those characters that are relevant for the locale. While the Unicode Collation Algorithm combines a default collation order for the entire Unicode character set with the ability to tailor for local conventions, subsets and tailorings still result in differences in behaviour.

### 4.4.1 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 2021 Language Specification (ECMA-262 12th Edition), as ES2021:

- Object Internal Methods and Internal Slots, as described in ES2021, 6.1.7.2.
- Algorithm conventions, as described in ES2021, 5.2, and the use of abstract operations, as described in ES2021, 7.1, 7.2, 7.3, 7.4.
- Internal Slots, as described in ES2021, 9.1.
- The List and Record Specification Type, as described in ES2021, 6.2.1.

**NOTE** As described in the ECMAScript Language Specification, algorithms are used to precisely specify the required semantics of ECMAScript constructs, but are not intended to imply the use of any specific implementation technique. Internal slots are used to define the semantics of object values, but are not part of the API. They are defined purely for expository purposes. An implementation of the API must behave as if it produced and operated upon internal slots in the manner described here.

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

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

### 5.1 Well-Known Intrinsic Objects

The following table extends the Well-Known Intrinsic Objects table defined in ES2021, 6.1.7.4.
Table 1 — Well-known Intrinsic Objects (Extensions)

<table>
<thead>
<tr>
<th>Intrinsic Name</th>
<th>Global Name</th>
<th>ECMAScript Language Association</th>
</tr>
</thead>
<tbody>
<tr>
<td>%Collator%</td>
<td>Intl.Collator</td>
<td>The Intl.Collator constructor (10.1)</td>
</tr>
<tr>
<td>%DateTimeFormat%</td>
<td>Intl.DateTimeFormat</td>
<td>The Intl.DateTimeFormat constructor (11.2).</td>
</tr>
<tr>
<td>%DisplayNames%</td>
<td>Intl.DisplayNames</td>
<td>The Intl.DisplayNames constructor (12.2).</td>
</tr>
<tr>
<td>%Intl%</td>
<td>Intl</td>
<td>The Intl object (8).</td>
</tr>
<tr>
<td>%ListFormat%</td>
<td>Intl.ListFormat</td>
<td>The Intl.ListFormat constructor (13.2).</td>
</tr>
<tr>
<td>%Locale%</td>
<td>Intl/locale</td>
<td>The Intl.Locale constructor (14.1).</td>
</tr>
<tr>
<td>%NumberFormat%</td>
<td>Intl.NumberFormat</td>
<td>The Intl.NumberFormat constructor (15.2)</td>
</tr>
<tr>
<td>%RelativeTimeFormat%</td>
<td>Intl.RelativeTimeFormat</td>
<td>The Intl.RelativeTimeFormat constructor (17.2).</td>
</tr>
</tbody>
</table>

6 Identification of Locales, Currencies, TimeZones, and Measurement Units

This clause describes the String values used in the ECMAScript 2021 Internationalization API Specification to identify locales, currencies, time zones, and measurement units.

6.1 Case Sensitivity and Case Mapping

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

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

6.2 Language Tags

The ECMAScript 2021 Internationalization API Specification identifies locales using Unicode BCP 47 locale identifiers as defined by Unicode Technical Standard #35 LDML § 3 Unicode Language and Locale Identifiers, which may include extensions such as the Unicode BCP 47 U Extension. Their canonical form is specified in Unicode Technical Standard #35 LDML § 3.2.1 Canonical Unicode Locale Identifiers.
Unicode BCP 47 locale identifiers are structurally valid when they match those syntactical formatting criteria of Unicode Technical Standard 35, section 3.2, but it is not required to validate them according to the Unicode validation data. All structurally valid language tags are valid for use with the APIs defined by this standard. However, the set of locales and thus language tags that an implementation supports with adequate localizations is implementation dependent. The constructors Collator, NumberFormat, DateTimeFormat, PluralRules, ListFormat, and DisplayNames map the language tags used in requests to locales supported by their respective implementations.

6.2.1 Unicode Locale Extension Sequences

This standard uses the term "Unicode locale extension sequence" - as described in Unicode locale extensions in UTS 35 Unicode Locale Identifier, section 3.2 - for any substring of a language tag that is not part of a private use subtag sequence, starts with a separator "-" and the singleton "u", and includes the maximum sequence of following non-singleton subtags and their preceding "-" separators.

6.2.2 IsStructurallyValidLanguageTag (locale)

The IsStructurallyValidLanguageTag abstract operation determines whether the locale argument (which must be a String value) is a language tag recognized by this specification. (It does not consider whether the language tag conveys any meaningful semantics, differentiate between aliased subtags and their preferred replacement subtags, or require canonical casing or subtag ordering.)

IsStructurallyValidLanguageTag returns true if all of the following conditions hold, false otherwise:

- locale can be generated from the EBNF grammar for unicode_locale_id in Unicode Technical Standard #35 LDML § 3.2 Unicode Locale Identifier;
- locale not use any of the backwards compatibility syntax described in Unicode Technical Standard #35 LDML § 3.3 BCP 47 Conformance;
- the unicode_language_id within locale contains no duplicate unicode_variant_subtag subtags; and
- if locale contains an extensions* component, that component
  - does not contain any other_extensions components with duplicate [alphanum-][TuUxX]
    subtags,
  - contains at most one unicode_locale_extensions component,
  - contains at most one transformed_extensions component, and
  - if a transformed_extensions component that contains a tlang component is present, then
    - the tlang component contains no duplicate unicode_variant_subtag subtags.

When evaluating each condition, terminal value characters in the grammar are interpreted as the corresponding ASCII code points. Subtags are duplicates if they are ASCII case-insensitively equivalent.

NOTE Every string for which this function returns true is both a "Unicode BCP 47 locale identifier", consistent with Unicode Technical Standard #35 LDML § 3.2 Unicode Locale Identifier and Unicode Technical Standard #35 LDML § 3.3 BCP 47 Conformance, and a valid BCP 47 language tag.

6.2.3 CanonicalizeUnicodeLocaleId (locale)

The CanonicalizeUnicodeLocaleId abstract operation returns the canonical and case-regularized form of the locale argument (which must be a String value for which IsStructurallyValidLanguageTag(locale) equals true). The following steps are taken:

1. Let localeId be the string locale after performing the algorithm to transform it to canonical syntax per Unicode Technical Standard #35 LDML § 3.2.1 Canonical Unicode Locale Identifiers. (The result is a Unicode BCP 47 locale identifier, in canonical syntax but not necessarily in canonical form.)
2. Let `localeId` be the string `localeId` after performing the algorithm to transform it to canonical form. (The result is a Unicode BCP 47 locale identifier, in both canonical syntax and canonical form.)

3. If `localeId` contains a substring `extension` that is a Unicode locale extension sequence, then
   a. Let `components` be ! `UnicodeExtensionComponents(extension)`.
   b. Let `attributes` be `components.[[Attributes]]`.
   c. Let `keywords` be `components.[[Keywords]]`.
   d. Let `newExtension` be "u".
   e. For each element `attr` of `attributes`, do
      i. Append "-" to `newExtension`.
      ii. Append `attr` to `newExtension`.
   f. For each `Record` `{ [[Key]], [[Value]] } keyword in `keywords`, do
      1. Append "-" to `newExtension`.
      2. Append `keyword.[[Key]]` to `newExtension`.
      3. If `keyword.[[Value]]` is not the empty String, then
         1. Append "-" to `newExtension`.
         2. Append `keyword.[[Value]]` to `newExtension`.
   g. Assert: `newExtension` is not equal to "u".
   h. Let `localeId` be `localeId` with the substring corresponding to `extension` replaced by the string `newExtension`.

4. Return `localeId`.

**NOTE** The third step of this algorithm 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.

### 6.2.4 DefaultLocale ()

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

### 6.3 Currency Codes

The ECMAScript 2021 Internationalization API Specification identifies currencies using 3-letter currency codes as defined by ISO 4217. Their canonical form is upper case.

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

#### 6.3.1 IsWellFormedCurrencyCode ( `currency` )

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

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

© Ecma International 2021
6.4 Time Zone Names

The ECMAScript 2021 Internationalization API Specification identifies time zones using the Zone and Link names of the IANA Time Zone Database. Their canonical form is the corresponding Zone name in the casing used in the IANA Time Zone Database.

All registered Zone and Link names are allowed. Implementations must recognize all such names, and use best available current and historical information about their offsets from UTC and their daylight saving time rules in calculations. However, the set of combinations of time zone name and language tag for which localized time zone names are available is implementation dependent.

6.4.1 IsValidTimeZoneName ( timeZone )

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

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

6.4.2 CanonicalizeTimeZoneName

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

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

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

6.4.3 DefaultTimeZone ( )

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

6.5 Measurement Unit Identifiers

The ECMAScript 2021 Internationalization API Specification identifies measurement units using a core unit identifier as defined by Unicode Technical Standard #35, Part 2, Section 6. Their canonical form is a string containing all lowercase letters with zero or more hyphens.

Only a limited set of core unit identifiers are allowed. An illegal core unit identifier results in a RangeError.
6.5.1 IsWellFormedUnitIdentifier (unitIdentifier)

The IsWellFormedUnitIdentifier abstract operation verifies that the unitIdentifier argument (which must be a String value) represents a well-formed core unit identifier as defined in UTS #35, Part 2, Section 6. In addition to obeying the UTS #35 core unit identifier syntax, unitIdentifier must be one of the identifiers sanctioned by UTS #35 or be a compound unit composed of two sanctioned simple units. The following steps are taken:

1. If the result of IsSanctionedSimpleUnitIdentifier(unitIdentifier) is true, then
   a. Return true.
2. If the substring "-per-" does not occur exactly once in unitIdentifier, then
   a. Return false.
3. Let numerator be the substring of unitIdentifier from the beginning to just before "-per-".
4. If the result of IsSanctionedSimpleUnitIdentifier(numerator) is false, then
   a. Return false.
5. Let denominator be the substring of unitIdentifier from just after "-per-" to the end.
6. If the result of IsSanctionedSimpleUnitIdentifier(denominator) is false, then
   a. Return false.
7. Return true.

6.5.2 IsSanctionedSimpleUnitIdentifier (unitIdentifier)

The IsSanctionedSimpleUnitIdentifier abstract operation verifies that the given core unit identifier is among the simple units sanctioned in the current version of the ECMAScript Internationalization API Specification, a subset of the Validity Data as described in UTS #35, Part 1, Section 3.11; the list may grow over time. As discussed in UTS #35, a simple unit is one that does not have a numerator and denominator. The following steps are taken:

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

Table 2 — Simple units sanctioned for use in ECMAScript

<table>
<thead>
<tr>
<th>Simple Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>acre</td>
</tr>
<tr>
<td>bit</td>
</tr>
<tr>
<td>byte</td>
</tr>
<tr>
<td>celsius</td>
</tr>
<tr>
<td>centimeter</td>
</tr>
<tr>
<td>day</td>
</tr>
<tr>
<td>degree</td>
</tr>
<tr>
<td>fahrenheit</td>
</tr>
<tr>
<td>fluid-ounce</td>
</tr>
<tr>
<td>foot</td>
</tr>
<tr>
<td>gallon</td>
</tr>
<tr>
<td>Simple Unit</td>
</tr>
<tr>
<td>--------------------------</td>
</tr>
<tr>
<td>gigabit</td>
</tr>
<tr>
<td>gigabyte</td>
</tr>
<tr>
<td>gram</td>
</tr>
<tr>
<td>hectare</td>
</tr>
<tr>
<td>hour</td>
</tr>
<tr>
<td>inch</td>
</tr>
<tr>
<td>kilobit</td>
</tr>
<tr>
<td>kilobyte</td>
</tr>
<tr>
<td>kilogram</td>
</tr>
<tr>
<td>kilometer</td>
</tr>
<tr>
<td>liter</td>
</tr>
<tr>
<td>megabit</td>
</tr>
<tr>
<td>megabyte</td>
</tr>
<tr>
<td>meter</td>
</tr>
<tr>
<td>mile</td>
</tr>
<tr>
<td>mile-scandinavian</td>
</tr>
<tr>
<td>milliliter</td>
</tr>
<tr>
<td>millimeter</td>
</tr>
<tr>
<td>millimeter</td>
</tr>
<tr>
<td>millisecond</td>
</tr>
<tr>
<td>minute</td>
</tr>
<tr>
<td>month</td>
</tr>
<tr>
<td>ounce</td>
</tr>
<tr>
<td>percent</td>
</tr>
<tr>
<td>petabyte</td>
</tr>
<tr>
<td>pound</td>
</tr>
<tr>
<td>second</td>
</tr>
<tr>
<td>stone</td>
</tr>
<tr>
<td>terabit</td>
</tr>
<tr>
<td>terabyte</td>
</tr>
</tbody>
</table>
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 2021 Language Specification, 12th Edition, clause 17, or successor.

8 The Intl Object

The Intl object is the %Intl% intrinsic object and the initial value of the "Intl" property of the globalobject. The Intl object is a single ordinary object.

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

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

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

8.1 Value Properties of the Intl Object

8.1.1 Intl[ @@toStringTag ]

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

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

8.2 Constructor Properties of the Intl Object

8.2.1 Intl.Collator (...)

See 10.

8.2.2 Intl.DateTimeFormat (...)

See 11.

8.2.3 Intl.DisplayNames (...)

See 12.
8.2.4 Intl.ListFormat (…)
See 13.

8.2.5 Intl.Locale (…)
See 14.

8.2.6 Intl.NumberFormat (…)
See 15.

8.2.7 Intl.PluralRules (…)
See 16.

8.2.8 Intl.RelativeTimeFormat (…)
See 17.

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

8.3 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).

9 Locale and Parameter Negotiation

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

9.1 Internal slots of Service Constructors


- [[AvailableLocales]] is a List that contains structurally valid (6.2.2) and canonicalized (6.2.3) Unicode BCP 47 locale identifiers identifying the locales for which the implementation provides the functionality of the constructed objects. Language tags on the list must not have a Unicode locale extension sequence. The list must include the value returned by the DefaultLocale abstract operation (6.2.4), and must not
include duplicates. Implementations must include in [[AvailableLocales]] locales that can serve as fallbacks in the algorithm used to resolve locales (see 9.2.7). For example, implementations that provide a "de-DE" locale must include a "de" locale that can serve as a fallback for requests such as "de-AT" and "de-CH". For locales that include a script subtag in addition to language and region, the corresponding locale without a script subtag must also be supported; that is, if an implementation recognizes "zh-Hant-TW", it is also expected to recognize "zh-TW". The ordering of the locales within [[AvailableLocales]] is irrelevant.

- [[RelevantExtensionKeys]] is a List of keys of the language tag extensions defined in Unicode Technical Standard 35 that are relevant for the functionality of the constructed objects.
- [[SortLocaleData]] and [[SearchLocaleData]] (for Intl.Collator) and [[LocaleData]] (for Intl.DateTimeFormat, Intl.DisplayNames, Intl.ListFormat, Intl.NumberFormat, Intl.PluralRules, and Intl.RelativeTimeFormat) are records that have fields for each locale contained in [[AvailableLocales]]. The value of each of these fields must be a record that has fields for each key contained in [[RelevantExtensionKeys]]. The value of each of these fields must be a non-empty list of those values defined in Unicode Technical Standard 35 for the given key that are supported by the implementation for the given locale, with the first element providing the default value.

NOTE For example, an implementation of DateTimeFormat might include the language tag "th" in its [[AvailableLocales]] internal slot, and must (according to 11.3.3) include the key "ca" in its [[RelevantExtensionKeys]] internal slot. For Thai, the "buddhist" calendar is usually the default, but an implementation might also support the calendars "gregory", "chinese", and "islamicc" for the locale "th". The[[LocaleData]] internal slot would therefore at least include `[[th]]: {[[ca]]: "buddhist", "gregory", "chinese", "islamicc"}`.

9.2 Abstract Operations

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

9.2.1 CanonicalizeLocaleList ( locales )

The abstract operation CanonicalizeLocaleList takes the following steps:

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

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

### 9.2.2 BestAvailableLocale (availableLocales, locale)

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

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

### 9.2.3 LookupMatcher (availableLocales, requestedLocales)

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

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

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

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

9.2.5  UnicodeExtensionComponents ( extension )

The UnicodeExtensionComponents abstract operation returns the attributes and keywords from extension, which must be a String value whose contents are a Unicode locale extension sequence. If a attribute or a keyword occurs multiple times in extension, only the first occurrence is returned. The following steps are taken:

1. Let attributes be a new empty List.
2. Let keywords be a new empty List.
3. Let keyword be undefined.
4. Let size be the length of extension.
5. Let k be 3.
6. Repeat, while k < size,
   a. Let e be StringIndexOf(extension, "\", k).
   b. If e = -1, let len be size - k; else let len be e - k.
   c. Let subtag be the String value equal to the substring of extension consisting of the code units at indices k (inclusive) through k + len (exclusive).
   d. If keyword is undefined and len ≠ 2, then
      i. If subtag is not an element of attributes, then
         1. Append subtag to attributes.
   e. Else if len = 2, then
      i. If keyword is not undefined and keywords does not contain an element whose[[Key]] is the same as keyword.\([[[Key]]]]\), then
         1. Append keyword to keywords.
      ii. Set keyword to the Record ( [[Key]]: subtag, [[Value]]: "\" ).
   f. Else,
      i. If keyword.\([[[Value]]]]\) is the empty String, then
         1. Set keyword.\([[[Value]]]]\) to subtag.
      ii. Else,
         1. Set keyword.\([[[Value]]]]\) to the string-concatenation of keyword.\([[[Value]]]]\), "\", and subtag.
   g. Let k be k + len + 1.
7. If keyword is not undefined and keywords does not contain an element whose [[Key]] is the same as keyword.\([[[Key]]]]\), then
   a. Append keyword to keywords.
8. Return the Record ( [[Attributes]]: attributes, [[Keywords]]: keywords ).

9.2.6  InsertUnicodeExtensionAndCanonicalize ( locale, extension )

The InsertUnicodeExtensionAndCanonicalize abstract operation inserts extension, which must be a Unicode locale extension sequence, into locale, which must be a String value with a structurally valid and canonicalized Unicode BCP 47 locale identifier. The following steps are taken:

The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar.
1. Assert: `locale` does not contain a substring that is a Unicode locale extension sequence.
2. Assert: `extension` is a Unicode locale extension sequence.
3. Assert: `tag` matches the `unicode_locale_id` production.
4. Let `privateIndex` be `StringIndexOf(locale, "-x-", 0).
5. If `privateIndex` = -1, then
   a. Let `locale` be the string-concatenation of `locale` and `extension`.
6. Else,
   a. Let `preExtension` be the substring of `locale` from position 0, inclusive, to position `privateIndex`, exclusive.
   b. Let `postExtension` be the substring of `locale` from position `privateIndex` to the end of the string.
   c. Let `locale` be the string-concatenation of `preExtension`, `extension`, and `postExtension`.
7. Assert: `IsStructurallyValidLanguageTag(locale)` is true.
8. Return `! CanonicalizeUnicodeLocaleId(locale)`.

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

The ResolveLocale abstract operation compares a BCP 47 language priority list `requestedLocales` against the locales in `availableLocales` and determines the best available language to meet the request. `availableLocales`, `requestedLocales`, and `relevantExtensionKeys` must be provided as **List** values, **options** and **localeData** as Records.

The following steps are taken:

1. Let `matcher` be `options.[[localeMatcher]]`.
2. If `matcher` is "lookup", then
   a. Let `r` be `LookupMatcher(availableLocales, requestedLocales)`.
3. Else,
   a. Let `r` be `BestFitMatcher(availableLocales, requestedLocales)`.
4. Let `foundLocale` be `r.[[locale]]`.
5. Let `result` be a new **Record**.
6. Set `result.[[dataLocale]]` to `foundLocale`.
7. If `r` has an `[extension]` field, then
   a. Let `components` be `! UnicodeExtensionComponents(r.[[extension]])`.
   b. Let `keywords` be `components.[[Keywords]]`.
8. Let `supportedExtension` be "-u".
9. For each element `key` of `relevantExtensionKeys`, do
   a. Let `foundLocaleData` be `localeData.[[<foundLocale>]]`.
   b. Assert: Type(`foundLocaleData`) is **Record**.
   c. Let `keyLocaleData` be `foundLocaleData.[[<key>]]`.
   d. Assert: Type(`keyLocaleData`) is **List**.
   e. Let `value` be `keyLocaleData[0]`.
   f. Assert: Type(`value`) is either String or Null.
   g. Let `supportedExtensionAddition` be "".
   h. If `r` has an `[extension]` field, then
      i. If `keywords` contains an element whose [[Key]] is the same as `key`, then
         1. Let `entry` be the element of `keywords` whose [[Key]] is the same as `key`.
         2. Let `requestedValue` be `entry.[[Value]]`.
         3. If `requestedValue` is not the empty String, then
            a. If `keyLocaleData` contains `requestedValue`, then
               i. Let `value` be `requestedValue`.
               ii. Let `supportedExtensionAddition` be the string-concatenation of "-", `key`, "-", and `value`.
         4. Else if `keyLocaleData` contains "true", then
            a. Let `value` be "true".
b. Let supportedExtensionAddition be the string-concatenation of "-" and key.

i. If options has a field [[<key>]], then
   i. Let optionsValue be options.[[<key>]].
   ii. Assert: Type(optionsValue) is either String, Undefined, or Null.
   iii. If Type(optionsValue) is String, then
       1. Let optionsValue be the string optionsValue after performing the algorithm steps to transform Unicode extension values to canonical syntax per Unicode Technical Standard #35 LDML § 3.2.1 Canonical Unicode Locale Identifiers, treating key as value and optionsValue as value productions.
       2. Let optionsValue be the string optionsValue after performing the algorithm steps to transform Unicode extension values with their canonical form per Technical Standard #35 LDML § 3.2.1 Canonical Unicode Locale Identifiers, treating key as value and optionsValue as value productions.
   iv. If keyLocaleData contains optionsValue, then
       1. If SameValue(optionsValue, value) is false, then
          a. Let value be optionsValue.
          b. Let supportedExtensionAddition be "".
       j. Set result.[[<key>]] to value.
       k. Append supportedExtensionAddition to supportedExtension.
   10. If the number of elements in supportedExtension is greater than 2, then
       a. Let foundLocale be InsertUnicodeExtensionAndCanonicalize(foundLocale, supportedExtension).
   11. Set result.[[locale]] to foundLocale.
   12. Return result.

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

9.2.8 LookupSupportedLocales ( availableLocales, requestedLocales )

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

1. Let subset be a new empty List.
2. For each element locale of requestedLocales, do
   a. Let noExtensionsLocale be the String value that is locale with all Unicode locale extension sequences removed.
   b. Let availableLocale be BestAvailableLocale(availableLocales, noExtensionsLocale).
   c. If availableLocale is not undefined, append locale to the end of subset.
3. Return subset.

9.2.9 BestFitSupportedLocales ( availableLocales, requestedLocales )

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

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

1. Set options to ? CoerceOptionsToObject(options).
3. If matcher is "best fit", then
   a. Let supportedLocales be BestFitSupportedLocales(availableLocales, requestedLocales).
4. Else,
   a. Let supportedLocales be LookupSupportedLocales(availableLocales, requestedLocales).
5. Return CreateArrayFromList(supportedLocales).

9.2.11 GetOptionsObject (options)

The abstract operation GetOptionsObject 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.

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

9.2.12 CoerceOptionsToObject (options)

The abstract operation CoerceOptionsToObject 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.

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

9.2.13 GetOption (options, property, type, values, fallback)

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

1. Assert: Type(options) is Object.
2. Let value be ? Get(options, property).
3. If value is undefined, return fallback.
4. Assert: type is "boolean" or "string".
5. If type is "boolean", then
   a. Let value be ! ToBoolean(value).
6. If type is "string", then
   a. Let value be ? ToString(value).
7. If values is not undefined and values does not contain an element equal to value, throw a RangeError exception.
8. Return value.
9.2.14 DefaultNumberOption ( value, minimum, maximum, fallback )

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

1. If value is undefined, return fallback.
2. Let value be ? ToNumber(value).
3. If value is NaN or less than minimum or greater than maximum, throw a RangeError exception.
4. Return floor(value).

9.2.15 GetNumberOption ( options, property, minimum, maximum, fallback )

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

1. Assert: Type(options) is Object.
2. Let value be ? Get(options, property).
3. Return ? DefaultNumberOption(value, minimum, maximum, fallback).

9.2.16 PartitionPattern ( pattern )

The PartitionPattern abstract operation is called with argument pattern. This abstract operation parses an abstract pattern string into a list of Records with two fields, [[Type]] and [[Value]]. The [[Value]] field will be a String value if [[Type]] is "literal", and undefined otherwise. The syntax of the abstract pattern strings is an implementation detail and is not exposed to users of ECMA-402. The following steps are taken:

1. Let result be a new empty List.
2. Let beginIndex be ⌈StringLengthOf(pattern, "\"\"", 0).
3. Let endIndex be 0.
4. Let nextIndex be 0.
5. Let length be the number of code units in pattern.
6. Repeat, while beginIndex is an integer index into pattern,
   a. Set endIndex to ⌈StringLengthOf(pattern, "]\"\"", beginIndex).
   b. Assert: endIndex is greater than beginIndex.
   c. If endIndex is greater than nextIndex, then
      i. Let literal be a substring of pattern from position nextIndex, inclusive, to position beginIndex, exclusive.
      ii. Append a new Record { [[Type]]: "literal", [[Value]]: literal } as the last element of the list result.
   d. Let p be the substring of pattern from position beginIndex, exclusive, to position endIndex, exclusive.
   e. Append a new Record { [[Type]]: p, [[Value]]: undefined } as the last element of the list result.
   f. Set nextIndex to endIndex + 1.
   g. Set beginIndex to ⌈StringLengthOf(pattern, "]\"\", nextIndex).
7. If nextIndex is less than length, then
   a. Let literal be the substring of pattern from position nextIndex, inclusive, to position length, exclusive.
   b. Append a new Record { [[Type]]: "literal", [[Value]]: literal } as the last element of the list result.
8. Return result.

10 Collator Objects

10.1 The Intl.Collator Constructor

The Intl.Collator constructor is the %Collator% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.
10.1.1 InitializeCollator (collator, locales, options)

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

The following algorithm refers to the type nonterminal from UTS 35’s Unicode Locale Identifier grammar.
1. Let requestedLocales be ? CanonicalizeLocaleList(locales).
2. Set options to ? CoerceOptionsToObject(options).
4. Set collator.[[Usage]] to usage.
5. If usage is "sort", then
   a. Let localeData be %Collator%.[[SortLocaleData]].
   b. Else, let localeData be %Collator%.[[SearchLocaleData]].
7. Let opt be a new Record.
9. Set opt.[[localeMatcher]] to matcher.
11. If collation is not undefined, then
   a. If collation does not match the Unicode Locale Identifier type nonterminal, throw a RangeError exception.
12. Set opt.[[co]] to collation.
14. If numeric is not undefined, then
   a. Let numeric be ! ToString(numeric).
15. Set opt.[[kn]] to numeric.
17. Set opt.[[kf]] to caseFirst.
18. Let relevantExtensionKeys be %Collator%.[[RelevantExtensionKeys]].
19. Let r be ResolveLocale(%Collator%.[[AvailableLocales]], requestedLocales, opt, relevantExtensionKeys, localeData).
20. Set collator.[[Locale]] to r.[[locale]].
21. Let collation be r.[[co]].
22. If collation is null, let collation be "default".
23. Set collator.[[Collation]] to collation.
24. If relevantExtensionKeys contains "kn", then
   a. Set collator.[[Numeric]] to ! SameValue(r.[[kn]], "true").
25. If relevantExtensionKeys contains "kf", then
   a. Set collator.[[CaseFirst]] to r.[[kf]].
27. If sensitivity is undefined, then
   a. If usage is "sort", then
      i. Let sensitivity be "variant".
   b. Else,
      i. Let dataLocale be r.[[dataLocale]].
      ii. Let dataLocaleData be localeData.[[<dataLocale>]].
      iii. Let sensitivity be dataLocaleData.[[sensitivity]].
28. Set collator.[[Sensitivity]] to sensitivity.
30. Set collator.[[IgnorePunctuation]] to ignorePunctuation.
31. Return collator.
10.1.2 Intl.Collator ([ locales [, options ] ])

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

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

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 %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 %Collator%.[[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).

The value of the "length" property of the supportedLocalesOf method is 1.

10.2.3 Internal Slots

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

NOTE  Unicode Technical Standard 35 describes ten locale extension keys that are relevant to collation: "co" for collator usage and specializations, "ka" for alternate handling, "kb" for backward second level weight, "kc" for case level, "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:
10.3 Properties of the Intl.Collator Prototype Object

The Intl.Collator prototype object is itself an ordinary object. `%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 the intrinsic object `%Collator%`.

10.3.2 Intl.Collator.prototype @@toStringTag

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

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

10.3.3 get Intl.Collator.prototype.compare

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

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

1. Let `collator` be the this value.
2. Perform ? RequireInternalSlot(`collator`, [[InitializedCollator]]).
3. If `collator`.[[BoundCompare]] is undefined, then
   a. Let `F` be a new built-in function object as defined in 10.3.3.1.
   b. Set `F`.[[Collator]] to `collator`.
   c. Set `collator`.[[BoundCompare]] to `F`.
4. Return `collator`.[[BoundCompare]].

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

10.3.3.1 Collator Compare Functions

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

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

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

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

10.3.3.2 CompareStrings ( collator, x, y )

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

The sensitivity of `collator` is interpreted as follows:

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

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

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

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

The CompareStrings abstract operation with any given `collator` argument, if considered as a function of the remaining two arguments `x` and `y`, must be a consistent comparison function (as defined in ES2021, 22.1.3.27) on the set of all Strings.

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

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

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

10.3.4 Intl.Collator.prototype.resolvedOptions ()

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

1. Let `collator` be 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 \( \text{collator}'s \) 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 \( \text{extensionKey} \) be the Extension Key value of the current row.
   ii. If \%Collator%.[[RelevantExtensionKeys]] does not contain \( \text{extensionKey} \), then
      1. Let \( v \) be \( \text{undefined} \).

d. If \( v \) is not \( \text{undefined} \), then
   i. Perform \(!\text{CreateDataPropertyOrThrow}(\text{options}, p, v)\).

5. Return \( \text{options} \).

### Table 3 — Resolved Options of Collator Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
<th>Extension Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Locale]]</td>
<td>&quot;locale&quot;</td>
<td></td>
</tr>
<tr>
<td>[[Usage]]</td>
<td>&quot;usage&quot;</td>
<td></td>
</tr>
<tr>
<td>[[Sensitivity]]</td>
<td>&quot;sensitivity&quot;</td>
<td></td>
</tr>
<tr>
<td>[[IgnorePunctuation]]</td>
<td>&quot;ignorePunctuation&quot;</td>
<td></td>
</tr>
<tr>
<td>[[Collation]]</td>
<td>&quot;collation&quot;</td>
<td></td>
</tr>
<tr>
<td>[[Numeric]]</td>
<td>&quot;numeric&quot;</td>
<td>&quot;kn&quot;</td>
</tr>
<tr>
<td>[[CaseFirst]]</td>
<td>&quot;caseFirst&quot;</td>
<td>&quot;kf&quot;</td>
</tr>
</tbody>
</table>

### 10.4 Properties of Intl.Collator Instances

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

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

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

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

Intl.Collator instances also have the following internal slots if the key corresponding to the name of the internal slot in Table 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 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>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Weekday]]</td>
<td>&quot;weekday&quot;</td>
<td>&quot;narrow&quot;, &quot;short&quot;, &quot;long&quot;</td>
</tr>
<tr>
<td>[[Era]]</td>
<td>&quot;era&quot;</td>
<td>&quot;narrow&quot;, &quot;short&quot;, &quot;long&quot;</td>
</tr>
<tr>
<td>[[Year]]</td>
<td>&quot;year&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;</td>
</tr>
<tr>
<td>[[Month]]</td>
<td>&quot;month&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;, &quot;narrow&quot;, &quot;short&quot;, &quot;long&quot;</td>
</tr>
<tr>
<td>[[Day]]</td>
<td>&quot;day&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;</td>
</tr>
<tr>
<td>[[DayPeriod]]</td>
<td>&quot;dayPeriod&quot;</td>
<td>&quot;narrow&quot;, &quot;short&quot;, &quot;long&quot;</td>
</tr>
<tr>
<td>[[Hour]]</td>
<td>&quot;hour&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;</td>
</tr>
<tr>
<td>[[Minute]]</td>
<td>&quot;minute&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;</td>
</tr>
<tr>
<td>[[Second]]</td>
<td>&quot;second&quot;</td>
<td>&quot;2-digit&quot;, &quot;numeric&quot;</td>
</tr>
<tr>
<td>[[FractionalSecondDigits]]</td>
<td>&quot;fractionalSecondDigits&quot;</td>
<td>1𝔽, 2𝔽, 3𝔽</td>
</tr>
<tr>
<td>[[TimeZoneName]]</td>
<td>&quot;timeZoneName&quot;</td>
<td>&quot;short&quot;, &quot;long&quot;</td>
</tr>
</tbody>
</table>

11.1.1 InitializeDateTimeFormat ( \texttt{dateTimeFormat, locales, options} )

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

The following algorithm refers to the \texttt{type} nonterminal from UTS 35's Unicode Locale Identifier grammar.

1. Let \texttt{requestedLocales} be \texttt{CanonicalizeLocaleList(locales)}.
2. Let \texttt{options} be \texttt{ToDateTimeOptions(options, "any", "date")}.
3. Let \texttt{opt} be a new Record.
4. Let \texttt{matcher} be \texttt{GetOption(options, "localeMatcher", "string", « "lookup", "best fit" », "best fit")}.
5. Set \texttt{opt.[localeMatcher]} to \texttt{matcher}.
6. Let \texttt{calendar} be \texttt{GetOption(options, "calendar", "string", undefined, undefined)}.
7. If \texttt{calendar} is not \texttt{undefined}, then
   a. If \texttt{calendar} does not match the Unicode Locale Identifier \texttt{type} nonterminal, throw a \texttt{RangeError} exception.
8. Set \texttt{opt.[calendar]} to \texttt{calendar}.
9. Let \texttt{numberingSystem} be \texttt{GetOption(options, "numberingSystem", "string", undefined, undefined)}.
10. If `numberingSystem` is not `undefined`, then
   a. If `numberingSystem` does not match the Unicode Locale Identifier type nonterminal, throw a
      `RangeError` exception.
11. Set `opt.[null]` to `numberingSystem`.
14. If `hour12` is not `undefined`, then
   a. Let `hourCycle` be `null`.
   b. Set `opt.[[hc]]` to `hourCycle`.
15. Let `localeData` be `%DateTimeFormat%.[[LocaleData]]`.
16. Let `r` be `ResolveLocale(%DateTimeFormat%.[[AvailableLocales]], requestedLocales, opt, %DateTimeFormat%.[[RelevantExtensionKeys]], localeData)`.
17. Set `dateTimeFormat`.[[Locale]] to `r.[[locale]]`.
18. Let `calendar` be `r.[[calendar]]`.
19. Set `dateTimeFormat`.[[Calendar]] to `calendar`.
20. Set `dateTimeFormat`.[[HourCycle]] to `r.[[hc]]`.
21. Set `dateTimeFormat`.[[NumberingSystem]] to `r.[[nu]]`.
22. Let `dataLocale` be `r.[[dataLocale]]`.
23. Let `timeZone` be ? `Get(options, "timeZone")`.
24. If `timeZone` is `undefined`, then
   a. Let `timeZone` be `DefaultValue()`.
25. Else,
   a. Let `timeZone` be ? `ToString(timeZone)`.
   b. If the result of `isValidTimeZoneName(timeZone)` is `false`, then
      i. Throw a `RangeError` exception.
   c. Let `timeZone` be `CanonicalizeTimeZoneName(timeZone)`.
26. Set `dateTimeFormat`.[[TimeZone]] to `timeZone`.
27. Let `opt` be a new `Record`.
28. For each row of `Table 4`, except the header row, in table order, do
   a. Let `prop` be the name given in the Property column of the row.
   b. If `prop` is "fractionalSecondDigits", then
      i. Let `value` be ? `GetNumberOption(options, "fractionalSecondDigits", 1, 3, undefined)`.
   c. Else,
      i. Let `value` be ? `GetOption(options, prop, "string", "the strings given in the Values column of the row", undefined)`.
   d. Set `opt.[[prop]]` to `value`.
29. Let `dataLocaleData` be `localeData.[[<dataLocale>]]`.
31. Let `dateStyle` be ? `GetOption(options, "dateStyle", "string", ""full", "long", "medium", "short", undefined)`.
32. Set `dateTimeFormat`.[[DateString]] to `dateStyle`.
34. Set `dateTimeFormat`.[[TimeFormat]] to `timeStyle`.
35. If `dateStyle` is not `undefined` or `timeStyle` is not `undefined`, then
   a. For each row in `Table 4`, except the header row, do
      i. Let `prop` be the name given in the Property column of the row.
      ii. Let `p` be `opt.[[prop]]`.
      iii. If `p` is not `undefined`, then
          1. Throw a `TypeError` exception.
   b. Let `styles` be `dataLocaleData.[[styles]]`.[[calendar]]
   c. Let `bestFormat` be `DateTimeStyleFormat(dateStyle, timeStyle, styles)`.
36. Else,
   a. Let `formats` be `dataLocaleData.[[formats]]`.[[calendar]]
   b. If `matcher` is "basic", then
i. Let \( \text{bestFormat} \) be \( \text{BasicFormatMatcher}(\text{opt}, \text{formats}) \).

38. For each row in Table 4, except the header row, in table order, do
   a. Let \( \text{prop} \) be the name given in the Property column of the row.
   b. If \( \text{bestFormat} \) has a field \( \langle \text{prop} \rangle \), then
      i. Let \( \text{p} \) be \( \text{bestFormat.}\langle \text{prop} \rangle \).
      ii. Set \( \text{dateTimeFormat} \)'s internal slot whose name is the Internal Slot column of the row to \( \text{p} \).
   c. Else,
      i. Let \( \text{bestFormat} \) be \( \text{BestFitFormatMatcher}(\text{opt}, \text{formats}) \).

39. If \( \text{dateTimeFormat.}\langle \text{Hour} \rangle \) is \( \text{undefined} \), then
   a. Set \( \text{dateTimeFormat.}\langle \text{HourCycle} \rangle \) to \( \text{undefined} \).
   b. Let \( \text{pattern} \) be \( \text{bestFormat.}\langle \text{pattern} \rangle \).
   c. Let \( \text{rangePatterns} \) be \( \text{bestFormat.}\langle \text{rangePatterns} \rangle \).

40. Else,
   a. Let \( \text{hcDefault} \) be \( \text{dataLocaleData.}\langle \text{hourCycle} \rangle \).
   b. Let \( \text{hc} \) be \( \text{dateTimeFormat.}\langle \text{HourCycle} \rangle \).
   c. If \( \text{hc} \) is \( \text{null} \), then
      i. Set \( \text{hc} \) to \( \text{hcDefault} \).
   d. If \( \text{hour12} \) is \( \text{undefined} \), then
      i. If \( \text{hour12} \) is \( \text{true} \), then
         1. If \( \text{hcDefault} \) is \( "h11" \) or \( "h23" \), then
            a. Set \( \text{hc} \) to \( "h11" \).
         2. Else,
            a. Set \( \text{hc} \) to \( "h12" \).
      ii. Else,
          1. Assert: \( \text{hour12} \) is \( \text{false} \).
          2. If \( \text{hcDefault} \) is \( "h11" \) or \( "h23" \), then
             a. Set \( \text{hc} \) to \( "h23" \).
          3. Else,
             a. Set \( \text{hc} \) to \( "h24" \).
   e. Set \( \text{dateTimeFormat.}\langle \text{HourCycle} \rangle \) to \( \text{hc} \).
   f. If \( \text{dateTimeFormat.}\langle \text{HourCycle} \rangle \) is \( "h11" \) or \( "h12" \), then
      i. Let \( \text{pattern} \) be \( \text{bestFormat.}\langle \text{pattern12} \rangle \).
      ii. Let \( \text{rangePatterns} \) be \( \text{bestFormat.}\langle \text{rangePatterns12} \rangle \).
   g. Else,
      i. Let \( \text{pattern} \) be \( \text{bestFormat.}\langle \text{pattern} \rangle \).
      ii. Let \( \text{rangePatterns} \) be \( \text{bestFormat.}\langle \text{rangePatterns} \rangle \).

41. Set \( \text{dateTimeFormat.}\langle \text{Pattern} \rangle \) to \( \text{pattern} \).
42. Set \( \text{dateTimeFormat.}\langle \text{RangePatterns} \rangle \) to \( \text{rangePatterns} \).
43. Return \( \text{dateTimeFormat} \).

11.1.2 ToDateTimeOptions ( \( \text{options, required, defaults} \) )

When the ToDateTimeOptions abstract operation is called with arguments \( \text{options, required, and defaults} \), the following steps are taken:

1. If \( \text{options} \) is \( \text{undefined} \), let \( \text{options} \) be \( \text{null} \); otherwise let \( \text{options} \) be \( ? \text{ToObject}(\text{options}) \).
2. Let \( \text{options} \) be \( \text{ OrdinaryObjectCreate}(\text{options}) \).
3. Let \( \text{needDefaults} \) be \( \text{true} \).
4. If \( \text{required} \) is \( \text{"date"} \) or \( \text{"any"} \), then
   a. For each of the property names \( \text{"weekday"}, \text{"year"}, \text{"month"}, \text{"day"}, \) do
      i. Let \( \text{prop} \) be the \( \text{property name} \).
      ii. Let \( \text{value} \) be \( ? \text{Get}(\text{options, prop}) \).
      iii. If \( \text{value} \) is not \( \text{undefined} \), let \( \text{needDefaults} \) be \( \text{false} \).
5. If \( \text{required} \) is \( \text{"time"} \) or \( \text{"any"} \), then
   a. For each of the property names \( \text{"dayPeriod"}, \text{"hour"}, \text{"minute"}, \text{"second"}, \text{"fractionalSecondDigits"}, \) do
11.1.3 DateTimeStyleFormat (dateStyle, timeStyle, styles)

The DateTimeStyleFormat abstract operation accepts arguments dateStyle and timeStyle, which are each either undefined, "full", "long", "medium", or "short", at least one of which is not undefined, and styles, which is a record from %DateTimeFormat% where [%locale%].[locale].[%styles%].calendar for some locale locale and calendar calendar. It returns the appropriate format record for date time formatting based on the parameters.

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

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

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

11.1.5 BestFitFormatMatcher (options, formats)

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

11.1.6 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 \( dt \) be \( F.[[DateTimeFormat]] \).
2. **Assert:** `Type( dtf )` is Object and `dtf` has an `[[InitializedDateTimeFormat]]` internal slot.

3. If `date` is not provided or is `undefined`, then
   a. Let `x` be `Call(%Date.now%, undefined).

4. Else,
   a. Let `x` be `ToNumber(date).

5. Return `FormatDateTime(dtf, x).

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

### 11.1.7 FormatDateTimePattern ( `dateFormatFormat`, `patternParts`, `x`, `rangeFormatOptions` )

The FormatDateTimePattern abstract operation is called with arguments `dateFormatFormat` (which must be an object initialized as a DateTimeFormat), `patternParts` (which is a list of Records as returned by `PartitionPattern`), `x` (which must be a `Number` value), and `rangeFormatOptions` (which is a range pattern `Record` as used in `[[rangePattern]]` or `undefined`), interprets `x` as a `time value` as specified in ES2021, 20.4.1.1, and creates the corresponding parts according `pattern` and to the effective locale and the formatting options of `dateFormatFormat` and `rangeFormatOptions`. The following steps are taken:

1. Let `x` be `TimeClip(x).

2. If `x` is `NaN`, throw a `RangeError` exception.

3. Let `locale` be `dateFormatFormat.[[Locale]]

4. Let `nfOptions` be ` OrdinaryObjectCreate(null).

5. Perform ! ` CreateDataPropertyOrThrow(nfOptions, "useGrouping", false).

6. Let `nf` be ? `Construct(%NumberFormat%, { locales: locale, nfOptions })

7. Let `nf2Options` be ` OrdinaryObjectCreate(null).

8. Perform ! ` CreateDataPropertyOrThrow(nf2Options, "minimumIntegerDigits", 2).

9. Perform ! ` CreateDataPropertyOrThrow(nf2Options, "useGrouping", false).

10. Let `nf3` be ? `Construct(%NumberFormat%, { locales: locale, nf3Options })

11. Let `fractionalSecondDigits` be `dateFormatFormat.[[FractionalSecondDigits]]

12. If `fractionalSecondDigits` is not `undefined`, then
   a. Let `nf3Options` be ` OrdinaryObjectCreate(null).

   b. Perform ! ` CreateDataPropertyOrThrow(nf3Options, "minimumIntegerDigits", fractionalSecondDigits).

   c. Perform ! ` CreateDataPropertyOrThrow(nf3Options, "useGrouping", false).

   d. Let `nf3Options` be ` OrdinaryObjectCreate(null).

   e. Let `tm` be `ToLocalTime(x, dateFormatFormat.[[Calendar]], dateFormatFormat.[[TimeZone]])

13. Let `result` be a new empty `List`.

14. For each `Record { [[Type]], [[Value]] }` `patternPart` in `patternParts`, do
   a. Let `p` be `patternPart.[[Type]]

   b. If `p` is "literal", then
      i. Append a new `Record { [[Type]]: "literal", [[Value]]: patternPart.[[Value]] }` as the last element of the list `result`.

   c. Else if `p` is equal to "fractionalSecondDigits", then
      i. Let `v` be `tm.[[Millisecond]]

      ii. Let `v` be `floor(v × 10^{ fractionalSecondDigits - 3 })

      iii. Let `fv` be `FormatNumeric(nf3, v)

      iv. Append a new `Record { [[Type]]: "fractionalSecond", [[Value]]: fv }` as the last element of the list `result`.

   d. Else if `p` is equal to "dayPeriod", then
      i. Let `f` be the value of `dateFormatFormat`'s internal slot whose name is the Internal Slot column of the matching row.

      ii. Let `fv` be a String value representing the day period of `tm` in the form given by `f`; the String value depends upon the implementation and the effective locale of `dateFormatFormat`.

      iii. Append a new `Record { [[Type]]: p, [[Value]]: fv }` as the last element of the list `result`.

   e. Else if `p` matches a Property column of the row in Table 4, then
      i. If `rangeFormatOptions` is not `undefined`, let `f` be the value of `rangeFormatOptions`'s field whose name matches `p`.
ii. Else, let \( f \) be the value of `dateTimeFormat`'s internal slot whose name is the Internal Slot column of the matching row.

iii. Let \( v \) be the value of `tm`'s field whose name is the Internal Slot column of the matching row.

iv. If \( p \) is "year" and \( v \leq 0 \), let \( v = 1 - v \).

v. If \( p \) is "month", increase \( v \) by 1.

vi. If \( p \) is "hour" and `dateTimeFormat.[[HourCycle]]` is "h11" or "h12", then
1. Let \( v = v \mod 12 \).
2. If \( v = 0 \) and `dateTimeFormat.[[HourCycle]]` is "h12", let \( v = 12 \).

vii. If \( p \) is "hour" and `dateTimeFormat.[[HourCycle]]` is "h24", then
1. If \( v = 0 \), let \( v = 24 \).

viii. If \( f \) is "numeric", then
1. Let \( fv \) be `FormatNumeric(nf, v)`.

ix. Else if \( f \) is "2-digit", then
1. Let \( fv \) be `FormatNumeric(nf2, v)`.
2. If the "length" property of \( fv \) is greater than 2, let \( lv \) be the substring of \( fv \) containing the last two characters.

x. Else if \( f \) is "narrow", "short", or "long", then let \( fv \) be a String value representing \( v \) in the form given by \( f \); the String value depends upon the implementation and the effective locale and calendar of `dateTimeFormat`. If \( p \) is "month" and `rangeFormatOptions` is `undefined`, then the String value may also depend on whether `dateTimeFormat.[[Day]]` is `undefined`. If \( p \) is "month" and `rangeFormatOptions` is not `undefined`, then the String value may also depend on whether `rangeFormatOptions.[[day]]` is `undefined`. If \( p \) is "timeZoneName", then the String value may also depend on the value of the `[[InDST]]` field of `tm`.

xi. If \( p \) is "era" and `rangeFormatOptions` is `undefined`, then the String value may also depend on whether `dateTimeFormat.[[Era]]` is `undefined`. If \( p \) is "era" and `rangeFormatOptions` is not `undefined`, then the String value may also depend on whether `rangeFormatOptions.[[era]]` is `undefined`. If the implementation does not have a localized representation of \( f \), then use the String value of \( v \) itself.

xii. Append a new `Record` ( `[[Type]]`: \( p \), `[[Value]]`: `fv` ) as the last element of the list `result`.

f. Else if \( p \) is equal to "am/pm", then
i. Let \( v \) be `tm.[[Hour]]`.
ii. If \( v \) is greater than 11, then
1. Let \( lv \) be an implementation and locale dependent String value representing "post meridiem".

iii. Else,
1. Let \( lv \) be an implementation and locale dependent String value representing "ante meridiem".

iv. Append a new `Record` ( `[[Type]]`: "dayPeriod", `[[Value]]`: `fv` ) as the last element of the list `result`.

g. Else if \( p \) is equal to "relatedYear", then
i. Let \( v \) be `tm.[[RelatedYear]]`.
ii. Let \( lv \) be `FormatNumeric(nf, v)`.
iii. Append a new `Record` ( `[[Type]]`: "relatedYear", `[[Value]]`: `fv` ) as the last element of the list `result`.

h. Else if \( p \) is equal to "yearName", then
i. Let \( v \) be `tm.[[YearName]]`.
ii. Let \( lv \) be an implementation and locale dependent String value representing \( v \).
iii. Append a new `Record` ( `[[Type]]`: "yearName", `[[Value]]`: `fv` ) as the last element of the list `result`.

i. Else,
1. Let \( unknown \) be an implementation-, locale-, and numbering system-dependent String based on \( x \) and \( p \).
2. Append a new `Record` ( `[[Type]]`: "unknown", `[[Value]]`: `unknown` ) as the last element of `result`.

16. Return `result`.

NOTE 1 It is recommended that implementations use the locale and calendar dependent strings provided by the Common Locale Data Repository (available at http://cldr.unicode.org), and use CLDR "abbreviated" strings for `dateTimeFormat"short"` strings, and CLDR "wide" strings for `dateTimeFormat"long"` strings.

NOTE 2 It is recommended that implementations use the time zone information of the IANA Time Zone Database.
11.1.8 PartitionDateTimePattern (dateTimeFormat, x)

The PartitionDateTimePattern abstract operation is called with arguments dateTimeFormat (which must be an object initialized as a DateTimeFormat) and x (which must be a Number value), interprets x as a time value as specified in ES2021, 20.4.1.1, and creates the corresponding parts according to the effective locale and the formatting options of dateTimeFormat. The following steps are taken:

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

11.1.9 FormatDateTime (dateTimeFormat, x)

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

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

11.1.10 FormatDateTimeToParts (dateTimeFormat, x)

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

1. Let parts be ? PartitionDateTimePattern(dateTimeFormat, x).  
2. Let result be ArrayCreate(0).  
3. Let n be 0.  
4. For each Record { [[Type]], [[Value]] } part in 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 ! CreateDataProperty(result, ! ToString(n), O).  
   e. Increment n by 1.  
5. Return result.

11.1.11 PartitionDateTimeRangePattern (dateTimeFormat, x, y)

The PartitionDateTimeRangePattern abstract operation is called with arguments dateTimeFormat (which must be an object initialized as a DateTimeFormat), x (which must be a Number value) and y (which must be a Number value), interprets x and y as time values as specified in ES2021, 20.4.1.1, and creates the corresponding parts according to the effective locale and the formatting options of dateTimeFormat. The following steps are taken:

1. Let x be TimeClip(x).  
2. If x is NaN, throw a RangeError exception.  
3. Let y be TimeClip(y).  
4. If y is NaN, throw a RangeError exception.  
5. If x is greater than y, throw a RangeError exception.  
6. Let tm1 be ToLocalTime(x, dateTimeFormat.\neighbors\Calendar\neighbors\}, dateTimeFormat.\neighbors\TimeZone\neighbors\}).  
7. Let tm2 be ToLocalTime(y, dateTimeFormat.\neighbors\Calendar\neighbors\}, dateTimeFormat.\neighbors\TimeZone\neighbors\}).  
8. Let rangePatterns be dateTimeFormat.\neighbors\RangePatterns\neighbors\].  
9. Let rangePattern be undefined.
10. Let `dateFieldsPracticallyEqual` be `true`.
11. Let `patternContainsLargerDateField` be `false`.
12. While `dateFieldsPracticallyEqual` is `true` and `patternContainsLargerDateField` is `false`, repeat for each row of Table 6 in order, except the header row:
   a. Let `fieldName` be the name given in the Range Pattern Field column of the row.
   b. If `rangePatterns` has a field `[[<fieldName>]]`, let `rp` be `rangePatterns.[[<fieldName>]]`; else let `rp` be `undefined`.
   c. If `rangePattern` is not `undefined` and `rp` is `undefined`, then
      i. Set `patternContainsLargerDateField` to `true`.
    d. Else,
       i. Let `rangePattern` be `rp`.
       ii. If `fieldName` is equal to `[[AmPm]]`, then
           1. Let `v1` be `tm1.[[Hour]]`.
           2. Let `v2` be `tm2.[[Hour]]`.
           3. If `v1` is greater than 11 and `v2` is less or equal than 11, or `v1` is less or equal than 11 and `v2` is greater than 11, then
              a. Set `dateFieldsPracticallyEqual` to `false`.
       iii. Else if `fieldName` is equal to `[[DayPeriod]]`, then
            1. Let `v1` be a String value representing the day period of `tm1`; the String value depends upon the implementation and the effective locale of `dateTimeFormat`.
            2. Let `v2` be a String value representing the day period of `tm2`; the String value depends upon the implementation and the effective locale of `dateTimeFormat`.
            3. If `v1` is not equal to `v2`, then
               a. Set `dateFieldsPracticallyEqual` to `false`.
       iv. Else if `fieldName` is equal to `[[FractionalSecondDigits]]`, then
            1. Let `fractionalSecondDigits` be `dateTimeFormat.[[FractionalSecondDigits]]`.
            2. If `fractionalSecondDigits` is `undefined`, then
               a. Set `fractionalSecondDigits` to 3.
            3. Let `v1` be `tm1.[[Millisecond]]`.
            4. Let `v2` be `tm2.[[Millisecond]]`.
            5. Let `v1` be `floor(v1 × 10(fractionalSecondDigits - 3))`.
            6. Let `v2` be `floor(v2 × 10(fractionalSecondDigits - 3))`.
            7. If `v1` is not equal to `v2`, then
               a. Set `dateFieldsPracticallyEqual` to `false`.
       v. Else,
          1. Let `v1` be `tm1.[[<fieldName>]]`.
          2. Let `v2` be `tm2.[[<fieldName>]]`.
          3. If `v1` is not equal to `v2`, then
             a. Set `dateFieldsPracticallyEqual` to `false`.
13. If `dateFieldsPracticallyEqual` is `true`, then
    a. Let `pattern` be `dateTimeFormat.[[Pattern]]`.
    b. Let `patternParts` be `PartitionPattern(pattern)`.
    c. For each `Record` (`[[Type]], [[Value]]`) `r` in `result`, do
       i. Set `r.[[Source]]` to "shared".
    d. Return `result`.
14. Let `result` be a new empty `List`.
15. If `rangePattern` is `undefined`, then
    a. Let `rangePattern` be `rangePatterns.[[Default]]`.
16. For each `Record` (`[[Pattern]], [[Source]]`) `rangePatternPart` in `rangePattern.[[PatternParts]]`, do
    a. Let `pattern` be `rangePatternPart.[[Pattern]]`.
    b. Let `source` be `rangePatternPart.[[Source]]`.
    c. If `source` is "startRange" or "shared", then
       i. Let `z` be `x`.

© Ecma International 2021
d. Else,
   i. Let z be y.

11.1.12 FormatDateTimeRange (dateTimeFormat, x, y)

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

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

11.1.13 FormatDateTimeRangeToParts (dateTimeFormat, x, y)

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

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

11.1.14 ToLocalTime (t, calendar, timeZone)

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

1. Assert: Type(t) is Number.
2. If calendar is "gregory", then
   a. Let timeZoneOffset be the value calculated according to LocalITZA(t, true) where the local time zone is replaced with timezone timeZone.
   b. Let tz be the time value t + timeZoneOffset.
   c. Return a record with fields calculated from tz according to Table 5.
3. Else,
   a. Return a record with the fields of Column 1 of Table 5 calculated from t for the given calendar and timeZone. The calculations should use best available information about the specified calendar and timeZone, including current and historical information about timezone offsets from UTC and daylight saving time rules.
### Table 5 — Record returned byToLocalTime

<table>
<thead>
<tr>
<th>Field</th>
<th>Value Calculation for Gregorian Calendar</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Weekday]]</td>
<td>WeekDay(tz) specified in ES2021’s Week Day</td>
</tr>
<tr>
<td>[[Era]]</td>
<td>Let year be YearFromTime(tz) specified in ES2021’s Year Number. If year is less than 0, return ‘BC’, else, return ‘AD’.</td>
</tr>
<tr>
<td>[[Year]]</td>
<td>YearFromTime(tz) specified in ES2021’s Year Number</td>
</tr>
<tr>
<td>[[RelatedYear]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[YearName]]</td>
<td>undefined</td>
</tr>
<tr>
<td>[[Month]]</td>
<td>MonthFromTime(tz) specified in ES2021’s Month Number</td>
</tr>
<tr>
<td>[[Day]]</td>
<td>DateFromTime(tz) specified in ES2021’s Date Number</td>
</tr>
<tr>
<td>[[Hour]]</td>
<td>HourFromTime(tz) specified in ES2021’s Hours, Minutes, Second, and Milliseconds</td>
</tr>
<tr>
<td>[[Minute]]</td>
<td>MinFromTime(tz) specified in ES2021’s Hours, Minutes, Second, and Milliseconds</td>
</tr>
<tr>
<td>[[Second]]</td>
<td>SecFromTime(tz) specified in ES2021’s Hours, Minutes, Second, and Milliseconds</td>
</tr>
<tr>
<td>[[Millisecond]]</td>
<td>msFromTime(tz) specified in ES2021’s Hours, Minutes, Second, and Milliseconds</td>
</tr>
<tr>
<td>[[InDST]]</td>
<td>Calculate true or false using the best available information about the specified calendar and timeZone, including current and historical information about time zone offsets from UTC and daylight saving time rules.</td>
</tr>
</tbody>
</table>

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

11.1.15 **UnwrapDateTimeFormat (dtf)**

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

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

**NORMATIVE OPTIONAL**

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

3. Perform ? RequireInternalSlot(dtf, [[InitializedDateTimeFormat]]).

4. Return dtf.

**NOTE** See 8.2 Note 1 for the motivation of the normative optional text.

11.2 **The Intl.DateTimeFormat Constructor**

The Intl.DateTimeFormat constructor is the %DateTimeFormat% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.19.1.
11.2.1 Intl.DateTimeFormat ([ locales [, options ] ])

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

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

NORMATIVE OPTIONAL

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

NOTE See 8.2 Note 1 for the motivation of the normative optional text.

11.3 Properties of the Intl.DateTimeFormat Constructor

The Intl.DateTimeFormat constructor has the following properties:

11.3.1 Intl.DateTimeFormat.prototype

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

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

11.3.2 Intl.DateTimeFormat.supportedLocalesOf ( locales [, options ] )

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

1. Let availableLocales be %DateTimeFormat%, [[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).

The value of the "length" property of the supportedLocalesOf method is 1.

11.3.3 Internal slots

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

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

NOTE Unicode Technical Standard 35 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","tradiio", or "finance".
- [[LocaleData]][[locale]].[[hc]] must be « null, "h11", "h12", "h23", "h24" ».
- [[LocaleData]][[locale]].[[hourCycle]] must be a String value equal to "h11", "h12", "h23", or "h24".
- [[LocaleData]][[locale]].[[formats]] must have a [[formats]] field. This [[formats]] field must be a Record with [[calendar]] fields for all calendar values calendar. The value of this field must be a list of records, each of which has a subset of the fields shown in Table 4, where each field must have one of the values specified for the field in Table 4. Multiple records in a list may use the same subset of the fields as long as they have different values for the fields. The following subsets must be available for each locale:
  - weekday, year, month, day, hour, minute, second, fractionalSecondDigits
  - weekday, year, month, day, hour, minute, second
  - weekday, year, month, day
  - year, month, day
  - year, month
  - month, day
  - hour, minute, second, fractionalSecondDigits
  - hour, minute, second
  - hour, minute
  - dayPeriod, hour
  - dayPeriod, hour, minute, second
  - dayPeriod, hour, minute

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

6. If the record has a `[[year]]` field, the `[rangePatterns]` and `[rangePatterns12]` fields may contain range patterns where the `[[Pattern]]` values may contain the substrings `"{yearName}"` and `"{relatedYear}"`.

   - `[localeData]` must have a `[[styles]]` field. The `[[styles]]` field must be a **Record** with `[[calendar]]` fields for all calendar values `calendar`. The calendar records must contain `[[DateTimeFormat]],` `[[TimeFormat]],` `[[DateTimeRangeFormat]]` and `[rangePatterns]` fields, the value of these fields are Records, where each of which has `[[full]],` `[[long]],` `[[medium]]` and `[[short]]` fields. For `[[DateTimeFormat]]` and `[rangePatterns]`, the value of these fields must be a record, which has a subset of the fields shown in **Table 4**, where each field must have one of the values specified for the field in **Table 4**. Each of the records must also have the following fields:

   1. A `[[pattern]]` field, whose value is a **String** value that contains for each of the date and time format component fields of the record a substring starting with `"{"`, followed by the name of the field, followed by `"}"`.
   2. If the record has an `[[hour]]` field, it must also have a `[[pattern12]]` field, whose value is a **String** value that, in addition to the substrings of the pattern field, contains at least one of the substrings `"{ampm}"` or `"{dayPeriod}"`.
   3. A `[rangePatterns]` field that contains a record similar to the one described in the `[formats]` field.
   4. If the record has an `[[hour]]` field, it must also have a `[[rangePatterns12]]` field. Its value is similar to the record in `[[rangePatterns]]` but it uses a string similar to `[[pattern12]]` for each range pattern.

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

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

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

- [[Minute]]:
  - [[RangePattern]]: "startRange", [[Pattern]]: "(hour):(minute)"
  - [[RangePattern]]: "shared", [[Pattern]]: "－"
  - [[RangePattern]]: "endRange", [[Pattern]]: "(hour):(minute)"
  - [[RangePattern]]: "shared", [[Pattern]]: "(ampm)"

- [[Default]]:
  - [[RangePattern]]: "startRange", [[Pattern]]: "(day)/(month)/(year), (hour):(minute) {ampm}"
  - [[RangePattern]]: "shared", [[Pattern]]: "－"
  - [[RangePattern]]: "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 [http://cldr.unicode.org](http://cldr.unicode.org)).

### Table 6 — Range pattern fields

<table>
<thead>
<tr>
<th>Range Pattern Field</th>
<th>Pattern String Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;era&quot;</td>
<td>[[Era]]</td>
</tr>
<tr>
<td>&quot;year&quot;</td>
<td>[[Year]]</td>
</tr>
<tr>
<td>&quot;month&quot;</td>
<td>[[Month]]</td>
</tr>
<tr>
<td>&quot;day&quot;</td>
<td>[[Day]]</td>
</tr>
<tr>
<td>&quot;ampm&quot;</td>
<td>[[AmPm]]</td>
</tr>
<tr>
<td>&quot;dayPeriod&quot;</td>
<td>[[DayPeriod]]</td>
</tr>
<tr>
<td>&quot;hour&quot;</td>
<td>[[Hour]]</td>
</tr>
<tr>
<td>&quot;minute&quot;</td>
<td>[[Minute]]</td>
</tr>
<tr>
<td>&quot;second&quot;</td>
<td>[[Second]]</td>
</tr>
<tr>
<td>&quot;fractionalSecondDigits&quot;</td>
<td>[[FractionalSecondDigits]]</td>
</tr>
</tbody>
</table>
11.4 Properties of the Intl.DateTimeFormat Prototype Object

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

11.4.1 Intl.DateTimeFormat.prototype.constructor

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

11.4.2 Intl.DateTimeFormat.prototype[@@toStringTag]

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

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

11.4.3 get Intl.DateTimeFormat.prototype.format

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

1. Let dtf be the this value.
2. If Type(dtf) is not Object, throw a TypeError exception.
3. Let dtf be UnwrapDateTimeFormat(dtf).
4. If dtf. [[BoundFormat]] is undefined, then
   a. Let F be a new built-in function object as defined in DateTime Format Functions (11.1.6).
   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.

11.4.4 Intl.DateTimeFormat.prototype.formatToParts (date)

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

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

11.4.5 Intl.DateTimeFormat.prototype.formatRange (startDate, endDate)

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

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

11.4.6 Intl.DateTimeFormat.prototype.formatRangeToParts (\( \text{startDate}, \text{endDate} \))

When the \( \text{formatRangeToParts} \) method is called with arguments \( \text{startDate} \) and \( \text{endDate} \), the following steps are taken:

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

11.4.7 Intl.DateTimeFormat.prototype.resolvedOptions ( )

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

1. Let \( \text{df} \) be the this value.
2. If Type(\( \text{df} \)) is not Object, throw a \text{TypeError} exception.
3. Let \( \text{df} \) be \( \text{UnwrapDateTimeFormat}(\text{df}) \).
4. Let \( \text{options} \) be \( \text{ OrdinaryObjectCreate}(\%\text{Object.prototype}\% ) \).
5. For each row of Table 7, except the header row, in table order, do
   a. Let \( p \) be the Property value of the current row.
      b. If \( p \) is "\text{hour12}" , then
         i. Let \( \text{hc} \) be \( \text{df}.[[\text{HourCycle}]] \).
         ii. If \( \text{hc} \) is "\text{h11}" or "\text{h12}" , let \( v \) be true.
         iii. Else if, \( \text{hc} \) is "\text{h23}" or "\text{h24}" , let \( v \) be false.
         iv. Else, let \( v \) be undefined.
      c. Else,
         i. Let \( v \) be the value of \( \text{df} \)'s internal slot whose name is the Internal Slot value of the current row.
     d. If the Internal Slot value of the current row is an Internal Slot value in Table 4, then
        i. If \( \text{df}.[[\text{DateStyle}]] \) is not undefined or \( \text{df}.[[\text{TimeStyle}]] \) is not undefined, then
           1. Let \( v \) be undefined.
     e. If \( v \) is not undefined, then
        i. Perform \( \text{CreateDataPropertyOrThrow}(\text{options}, p, v) \).
6. Return \( \text{options} \).

### Table 7 — Resolved Options of DateTimeFormat Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Locale]]</td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td>[[Calendar]]</td>
<td>&quot;calendar&quot;</td>
</tr>
<tr>
<td>[[NumberingSystem]]</td>
<td>&quot;numberingSystem&quot;</td>
</tr>
<tr>
<td>[[TimeZone]]</td>
<td>&quot;timeZone&quot;</td>
</tr>
<tr>
<td>[[HourCycle]]</td>
<td>&quot;hourCycle&quot;</td>
</tr>
<tr>
<td></td>
<td>&quot;hour12&quot;</td>
</tr>
<tr>
<td>Internal Slot</td>
<td>Property</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>[[Weekday]]</td>
<td>&quot;weekday&quot;</td>
</tr>
<tr>
<td>[[Era]]</td>
<td>&quot;era&quot;</td>
</tr>
<tr>
<td>[[Year]]</td>
<td>&quot;year&quot;</td>
</tr>
<tr>
<td>[[Month]]</td>
<td>&quot;month&quot;</td>
</tr>
<tr>
<td>[[Day]]</td>
<td>&quot;day&quot;</td>
</tr>
<tr>
<td>[[DayPeriod]]</td>
<td>&quot;dayPeriod&quot;</td>
</tr>
<tr>
<td>[[Hour]]</td>
<td>&quot;hour&quot;</td>
</tr>
<tr>
<td>[[Minute]]</td>
<td>&quot;minute&quot;</td>
</tr>
<tr>
<td>[[Second]]</td>
<td>&quot;second&quot;</td>
</tr>
<tr>
<td>[[FractionalSecondDigits]]</td>
<td>&quot;fractionalSecondDigits&quot;</td>
</tr>
<tr>
<td>[[TimeZoneName]]</td>
<td>&quot;timeZoneName&quot;</td>
</tr>
<tr>
<td>[[DateStyle]]</td>
<td>&quot;dateStyle&quot;</td>
</tr>
<tr>
<td>[[TimeStyle]]</td>
<td>&quot;timeStyle&quot;</td>
</tr>
</tbody>
</table>

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

NOTE 1 In this version of the ECMAScript 2021 Internationalization API, the "timeZone" property will be the name of the default time zone if no "timeZone" property was provided in the options object provided to the Intl.DateTimeFormat constructor. The first edition left the "timeZone" property undefined in this case.

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

11.5 Properties of Intl.DateTimeFormat Instances

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

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

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

- [[Locale]] is a String value with the language tag of the locale whose localization is used for formatting.
- [[Calendar]] is a String value with the "type" given in Unicode Technical Standard 35 for the calendar used for formatting.
- [[NumberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
- [[TimeZone]] is a String value with the IANA time zone name of the time zone used for formatting.
- [[Weekday]], [[Era]], [[Year]], [[Month]], [[Day]], [[DayPeriod]], [[Hour]], [[Minute]], [[Second]], [[TimeZoneName]] are each either undefined, indicating that the component is not used for formatting, or one of the String values given in Table 4, indicating how the component should be presented in the
formatted output.

- `[[FractionalSecondDigits]]` is either `undefined` or a positive, non-zero integer `Number value` indicating the fraction digits to be used for fractional seconds. Numbers will be rounded or padded with trailing zeroes if necessary.
- `[[HourCycle]]` is a String value indicating whether the 12-hour format ("h11", "h12") or the 24-hour format ("h23", "h24") should be used. "h11" and "h23" start with hour 0 and go up to 11 and 23 respectively. "h12" and "h24" start with hour 1 and go up to 12 and 24. `[[HourCycle]]` is only used when `[[Hour]]` is not `undefined`.
- `[[DateStyle]], [[TimeStyle]]` are either `undefined`, or a String value with values "full", "long", "medium", or "short".
- `[[Pattern]]` is a String value as described in 11.3.3.
- `[[RangePatterns]]` is a `Record` as described in 11.3.3.

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

## 12 DisplayNames Objects

### 12.1 Abstract Operations for DisplayNames Objects

#### 12.1.1 CanonicalCodeForDisplayNames ( type, code )

The `CanonicalCodeForDisplayNames` abstract operation is called with arguments `type` and `code`. It verifies that the `code` argument represents a well-formed code according to the `type` argument and returns the case-regularized form of the `code`. The algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. The following steps are taken:

1. If `type` is "language", then
   a. If `code` does not match the `unicode_language_id` production, throw a `RangeError` exception.
   b. If `IsStructurallyValidLanguageTag(code)` is false, throw a `RangeError` exception.
   c. Set `code` to `CanonicalizeUnicodeLocaleId(code)`.
   d. Return `code`.
2. If `type` is "region", then
   a. If `code` does not match the `unicode_region_subtag` production, throw a `RangeError` exception.
   b. Let `code` be the result of mapping `code` to upper case as described in 6.1.
   c. Return `code`.
3. If `type` is "script", then
   a. If `code` does not match the `unicode_script_subtag` production, throw a `RangeError` exception.
   b. Let `code` be the result of mapping the first character in `code` to upper case, and mapping the second, third, and fourth character in `code` to lower case, as described in 6.1.
   c. Return `code`.
4. Assert: `type` is "currency".
5. If `IsWellFormedCurrencyCode(code)` is `false`, throw a `RangeError` exception.
6. Let `code` be the result of mapping `code` to upper case as described in 6.1.
7. Return `code`.

### 12.2 The Intl.DisplayNames Constructor

The `DisplayNames` constructor is the `%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.
12.2.1 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 %DisplayNames.prototype%, « [[InitializedDisplayNames]], [[Locale]], [[Style]], [[Type]], [[Fallback]], [[Fields]] ».
3. Let requestedLocales be CanonicalizeLocaleList(locales).
4. If options is undefined, throw a TypeError exception.
5. Let options be GetOptionsObject(options).
6. Let opt be a new Record.
7. Let localeData be %DisplayNames%.[[LocaleData]].
9. Set opt.[[localeMatcher]] to matcher.
10. Let r be ResolveLocale%DisplayNames%.[[AvailableLocales]], requestedLocales, opt, %DisplayNames%.[[RelevantExtensionKeys]]).
12. Set displayNames.[[Style]] to style.
14. If type is undefined, throw a TypeError exception.
15. Set displayNames.[[Type]] to type.
16. Let fallback be GetOption(options, "fallback", "string", « "code", "none" », "code").
17. Set displayNames.[[Fallback]] to fallback.
18. Set displayNames.[[Locale]] to the value of r.[[Locale]].
19. Let dataLocale be r.[[dataLocale]].
20. Let dataLocaleData be localeData.[[<dataLocale>]].
21. Let types be dataLocaleData.[[<types>]].
22. Assert: types is a Record (see 12.3.3).
23. Let typeFields be types.[[<type>]].
24. Assert: typeFields is a Record (see 12.3.3).
25. Let styleFields be typeFields.[[<style>]].
26. Assert: styleFields is a Record (see 12.3.3).
27. Set displayNames.[[Fields]] to styleFields.
28. Return displayNames.

12.3 Properties of the Intl.DisplayNames Constructor

The Intl.DisplayNames constructor has the following properties:

12.3.1 Intl.DisplayNames.prototype

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

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

12.3.2 Intl.DisplayNames.supportedLocalesOf ( locales [, options ] )

When the supportedLocalesOf method of %DisplayNames% is called, the following steps are taken:

1. Let availableLocales be %DisplayNames%.[[AvailableLocales]].
2. Let requestedLocales be CanonicalizeLocaleList(locales).
3. Return SupportedLocales(availableLocales, requestedLocales, options).
### 12.3.3 Internal slots

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

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

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

- `[[LocaleData]]` 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", and "currency".
- The value of the fields "language", "region", "script", and "currency" 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 matching the `unicode_language_id` production. 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.

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

### 12.4 Properties of the Intl.DisplayNames Prototype Object

The Intl.DisplayNames prototype object is itself an ordinary object. `%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.

#### 12.4.1 Intl.DisplayNames.prototype.constructor

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

#### 12.4.2 Intl.DisplayNames.prototype[ @@toStringTag ]

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

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

#### 12.4.3 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. Let `code` be `? 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`.

12.4.4 Intl.DisplayNames.prototype.resolvedOptions()

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

1. Let `displayNames` be `this` value.
2. Perform 𝒮RequireInternalSlot(`displayNames`, `[[InitializedDisplayNames]]`).
3. Let `options` be `! OrdinaryObjectCreate(%ObjectPrototype%)`.
4. For each row of Table 8, except the header row, in table order, do
   a. Let `p` be the Property value of the current row.
   b. Let `v` be the value of `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 8 — Resolved Options of DisplayNames Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[Locale]</code></td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td><code>[Style]</code></td>
<td>&quot;style&quot;</td>
</tr>
<tr>
<td><code>[Type]</code></td>
<td>&quot;type&quot;</td>
</tr>
<tr>
<td><code>[Fallback]</code></td>
<td>&quot;fallback&quot;</td>
</tr>
</tbody>
</table>

12.5 Properties of Intl.DisplayNames Instances

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

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

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

- `[Locale]` is a String value with the language tag of the locale whose localization is used for formatting.
- `[Style]` is one of the String values "narrow", "short", or "long", identifying the display name style used.
- `[Type]` is one of the String values "language", "region", "script", or "currency", 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.
- `[Fields]` is a Record (see 12.3.3) which must have fields with keys corresponding to codes according to `[Style]` and `[Type]`. 
13 ListFormat Objects

13.1 Abstract Operations for ListFormat Objects

13.1.1 DeconstructPattern (pattern, placeables)

The DeconstructPattern abstract operation is called with arguments pattern (which must be a String) and placeables (which must be a Record), and deconstructs the pattern string into a list of parts.

The placeables record is a record whose keys are placeables tokens used in the pattern string, and values are parts records 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"}
»

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

13.1.2 CreatePartsFromList (listFormat, list)

The CreatePartsFromList abstract operation is called with arguments listFormat (which must be an object initialized as a ListFormat) and list (which must be a List of String values), and creates the corresponding list of parts according to the effective locale and the formatting options of listFormat. Each part is a Record with two fields: [[Type]], which must be a string with values "element" or "literal", and [[Value]] which must be a string or a number. The following steps are taken:

1. Let size be the number of elements of list.
2. If size is 0, then
   a. Return a new empty List.
3. If size is 2, then
   a. Let n be an index into listFormat.[[Templates]] based on listFormat.[[Locale]], list[0], and list[1].
   b. Let pattern be listFormat.[[Templates]][n].[[Pair]].
c. Let first be a new Record \{ [[Type]]: "element", [[Value]]: list[0] \}.
d. Let second be a new Record \{ [[Type]]: "element", [[Value]]: list[1] \}.
e. Let placeables be a new Record \{ [[0]]: first, [[1]]: second \}.
f. Return DeconstructPattern(pattern, placeables).

4. Let last be a new Record \{ [[Type]]: "element", [[Value]]: list[size - 1] \}.

5. Let parts be « last ».
7. Repeat, while i ≥ 0,
   a. Let head be a new Record \{ [[Type]]: "element", [[Value]]: list[i] \}.
   b. Let n be an implementation-defined index into listFormat.[[Templates]] based on listFormat.[[Locale]], head, and parts.
   c. If i is 0, then
      i. Let pattern be listFormat.[[Templates]][n].[[Start]].
   d. Else if i is less than size - 2, then
      i. Let pattern be listFormat.[[Templates]][n].[[Middle]].
   e. Else,
      i. Let pattern be listFormat.[[Templates]][n].[[End]].
   f. Let placeables be a new Record \{ [[0]]: head, [[1]]: parts \}.
   g. Set parts to DeconstructPattern(pattern, placeables).
   h. Decrement i by 1.
8. Return parts.

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

13.1.3 FormatList ( listFormat, list )

The FormatList abstract operation is called with arguments listFormat (which must be an object initialized as a ListFormat) and list (which must be a List of String values), and performs the following steps:

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

13.1.4 FormatListToParts ( listFormat, list )

The FormatListToParts abstract operation is called with arguments listFormat (which must be an object initialized as a ListFormat) and list (which must be a List of String values), and performs the following steps:

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 in 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(n), O).
   e. Increment n by 1.
5. Return result.

13.1.5 StringListFromIterable ( iterable )

The abstract operation StringListFromIterable performs the following steps:
1. If `iterable` is `undefined`, then
   a. Return a new empty `List`.
2. Let `iteratorRecord` be ? `GetIterator(iterable)`.
3. Let `list` be a new empty `List`.
4. Let `next` be `true`.
5. Repeat, while `next` is not `false`,
   a. Set `next` to ? `IteratorStep(iteratorRecord)`.
   b. If `next` is not `false`, then
      i. Let `nextValue` be ? `IteratorValue(next)`.
      ii. If `Type(nextValue)` is not String, then
         1. Let `error` be `ThrowCompletion`(a newly created `TypeError` object).
         2. Return ? `IteratorClose(iteratorRecord, error)`.
      iii. Append `nextValue` to the end of the `List list`.
6. Return `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.

13.2 The Intl.ListFormat Constructor

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

13.2.1 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, "%ListFormat.prototype%", "`, `%ListFormat.prototype.prototype%", `[[InitializedListFormat]], [[Locale]], [[Style]], [[Templates]] `)`.
3. Let `requestedLocales` be ? `CanonicalizeLocaleList(locales)`.
4. Let `options` be ? `GetOptionsObject(options)`.
5. Let `opt` be a new `Record`.
7. Set `opt.[[localeMatcher]]` to `matcher`.
8. Let `localeData` be `%ListFormat%.[[LocaleData]]
9. Let `r` be `ResolveLocale(%ListFormat%, [[AvailableLocales]], `requestedLocales`, opt, `%ListFormat%`, `[[RelevantExtensionKeys]], localeData]`.
10. Set `listFormat.[[Locale]]` to `r.[[locale]]`.
11. Let `type` be ? `GetOption(options, "type", "string", ""conjunction", "disjunction", "unit" », "conjunction")`.
12. Set `listFormat.[[Type]]` to `type`.
14. Set `listFormat.[[Style]]` to `style`.
15. Let `dataLocale` be `r.[[dataLocale]]`
16. Let `dataLocaleData` be `localeData.[[<dataLocale>]]`.
17. Let `dataLocaleTypes` be `dataLocaleData.[[[type]]]`.
18. Set `listFormat.[[Templates]]` to `dataLocaleTypes.[[[style]]]`.
19. Return `listFormat`.

13.3 Properties of the Intl.ListFormat Constructor

The Intl.ListFormat constructor has the following properties:
13.3.1 Intl.ListFormat.prototype

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

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

13.3.2 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 %ListFormat%.[[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).

13.3.3 Internal slots

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

The value of the [[relevantExtensionKeys]] internal slot is "".

NOTE 1 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 %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 http://cldr.unicode.org/). In LDML's listPattern, conjunction corresponds to "standard", disjunction corresponds to "or", and unit corresponds to "unit".

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

13.4 Properties of the Intl.ListFormat Prototype Object

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

13.4.1 Intl.ListFormat.prototype.constructor

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

13.4.2 Intl.ListFormat.prototype [@@toStringTag]

The initial value of the @@toStringTag property is the String value "Intl.ListFormat".
This property has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: true }.

13.4.3 Intl.ListFormat.prototype.format ( list )

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

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

13.4.4 Intl.ListFormat.prototype.formatToParts ( list )

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

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

13.4.5 Intl.ListFormat.prototype.resolvedOptions ()

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

1. Let lf be the this value.
2. Perform ? RequireInternalSlot(lf, [[InitializedListFormat]]).
3. Let options be ! OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 9, except the header row, in table order, do
   a. Let p be the Property value of the current row.
   b. Let v be the value of lf's internal slot whose name is the Internal Slot value of the current row.
   c. Assert: v is not undefined.
   d. Perform ! CreateDataPropertyOrThrow(options, p, v).
5. Return options.

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Locale]]</td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td>[[Type]]</td>
<td>&quot;type&quot;</td>
</tr>
<tr>
<td>[[Style]]</td>
<td>&quot;style&quot;</td>
</tr>
</tbody>
</table>

13.5 Properties of Intl.ListFormat Instances

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

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

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

- [[Locale]] is a String value with the language tag of the locale whose localization is used by the list format
styles.
- [[Type]] is one of the String values "conjunction", "disjunction", or "unit", identifying the list of types used.
- [[Style]] is one of the String values "long", "short", or "narrow", identifying the list formatting style used.
- [[Templates]] is a ListFormat template set.

# 14 Locale Objects

## 14.1 The Intl.Locales Constructor

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

### 14.1.1 ApplyOptionsToTag ( tag, options )

The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar.

1. Assert: Type(tag) is String.
2. Assert: Type(options) is Object.
3. If IsStructurallyValidLanguageTag(tag) is false, throw a RangeError exception.
5. If language is not undefined, then
   a. If language does not match the unicode_language_subtag production, throw a RangeError exception.
6. If script is not undefined, then
   a. If script does not match the unicode_script_subtag production, throw a RangeError exception.
8. If region is not undefined, then
   a. If region does not match the unicode_region_subtag production, throw a RangeError exception.
9. Set tag to CanonicalizeUnicodeLocaleId(tag).
10. Assert: tag matches the unicode_locale_id production.
11. Let languageld be the substring of tag corresponding to the unicode_language_id production.
12. If language is not undefined, then
    a. Set languageld to languageld with the substring corresponding to the unicode_language_subtag production replaced by the string language.
13. If script is not undefined, then
    a. If languageld does not contain a unicode_script_subtag production, then
      i. Set languageld to the string-concatenation of the unicode_language_subtag production of languageld, ".", script, and the rest of languageld.
    b. Else,
      i. Set languageld to languageld with the substring corresponding to the unicode_script_subtag production replaced by the string script.
14. If region is not undefined, then
    a. If languageld does not contain a unicode_region_subtag production, then
      i. Set languageld to the string-concatenation of the unicode_language_subtag production of languageld, the substring corresponding to ".", and the "unicode_script_subtag" production if present, ".", region, and the rest of languageld.
    b. Else,
      i. Set languageld to languageld with the substring corresponding to the unicode_region_subtag production replaced by the string region.
15. Set tag to tag with the substring corresponding to the unicode_language_id production replaced by the string languageld.
16. Return CanonicalizeUnicodeLocaleId(tag).
14.1.2 ApplyUnicodeExtensionToTag (tag, options, relevantExtensionKeys)

The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar.

1. Assert: Type(tag) is String.
2. Assert: tag matches the unicode.locale_id production.
3. If tag contains a substring that is a Unicode locale extension sequence, then
   a. Let extension be the String value consisting of the first substring of tag that is a Unicode locale extension sequence.
   b. Let components be !UnicodeExtensionComponents( extension).
   c. Let attributes be components.[[Attributes]].
   d. Let keywords be components.[[Keywords]].
4. Else,
   a. Let attributes be a new empty List.
   b. Let keywords be a new empty List.
5. Let result be a new Record.
6. For each element key of relevantExtensionKeys, do
   a. Let value be undefined.
   b. If keywords contains an element whose [[Key]] is the same as key, then
      i. Let entry be the element of keywords whose [[Key]] is the same as key.
      ii. Let value be entry.[[Value]].
   c. Else,
      i. Let entry be empty.
   d. Assert: options has a field [[<_key_>]].
   e. Let optionsValue be options.[[<_key_>]].
   f. If optionsValue is not undefined, then
      i. Assert: Type(optionsValue) is String.
      ii. Let value be optionsValue.
      iii. If entry is not empty, then
            1. Set entry.[[Value]] to value.
      iv. Else,
            1. Append the Record { [[Key]]: key, [[Value]]: value } to keywords.
6. Set result.[[<_key_>]] to value.
7. Let locale be the String value that is tag with all Unicode locale extension sequences removed.
8. Let newExtension be a Unicode BCP 47 U Extension based on attributes and keywords.
9. If newExtension is not the empty String, then
   a. Let locale be !InsertUnicodeExtensionAndCanonicalize(locale, newExtension).
10. Set result.[[locale]] to locale.
11. Return result.

14.1.3 Intl.Locale (tag [, options])

The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. When the Intl.Locale function is called with an argument tag and an optional argument options, the following steps are taken:

1. If NewTarget is undefined, throw a TypeError exception.
2. Let relevantExtensionKeys be %Locale%.[[ RelevantExtensionKeys]].
3. Let internalSlotsList be « [[InitializedLocale]], [[Locale]], [[Calendar]], [[Collation]], [[HourCycle]],
   [[NumberingSystem]] ».
4. If relevantExtensionKeys contains "kf", then
   a. Append [[CaseFirst]] as the last element of internalSlotsList.
5. If relevantExtensionKeys contains "kn", then
   a. Append [[Numeric]] as the last element of internalSlotsList.
7. If Type(tag) is not String or Object, throw a TypeError exception.
8. If `Type(tag)` is Object and `tag` has an `[[InitializedLocale]]` internal slot, then
   a. Let `tag` be `tag.([Locale])`.
9. Else,
   a. Let `tag` be `toString(tag)`.
10. Set `options` to `CoerceOptionsToObject(options)`.
11. Set `tag` to `ApplyOptionsToTag(tag, options)`.
12. Let `opt` be a new Record.
13. Let `calendar` be `GetOption(options, "calendar", "string", undefined, undefined)`.
14. If `calendar` is not `undefined`, then
   a. If `calendar` does not match the `type` sequence (from UTS 35 Unicode Locale Identifier, section 3.2),
      throw a `RangeError` exception.
15. Set `opt.[[ca]]` to `calendar`.
16. Let `collation` be `GetOption(options, "collation", "string", undefined, undefined)`.
17. If `collation` is not `undefined`, then
   a. If `collation` does not match the `type` sequence (from UTS 35 Unicode Locale Identifier, section 3.2),
      throw a `RangeError` exception.
18. Set `opt.[[co]]` to `collation`.
20. Set `opt.[[hc]]` to `hc`.
22. Set `opt.[[kf]]` to `kf`.
23. Let `kn` be `GetOption(options, "numeric", "boolean", undefined, undefined)`.
24. If `kn` is not `undefined`, set `kn` to `toString(kn)`.
25. Set `opt.[[kn]]` to `kn`.
26. Let `numberingSystem` be `GetOption(options, "numberingSystem", "string", undefined, undefined)`.
27. If `numberingSystem` is not `undefined`, then
   a. If `numberingSystem` does not match the `type` sequence (from UTS 35 Unicode Locale Identifier, section 3.2),
      throw a `RangeError` exception.
28. Set `opt.[[nu]]` to `numberingSystem`.
29. Let `r` be `ApplyUnicodeExtensionToTag(tag, opt, relevantExtensionKeys)`.
30. Set `locale.([Locale])` to `r.[locale]`.
31. Set `locale.([Calendar])` to `r.[ca]`.
32. Set `locale.([Collation])` to `r.[co]`.
33. Set `locale.([HourCycle])` to `r.[hc]`.
34. If `relevantExtensionKeys` contains "kf", then
   a. Set `locale.([CaseFirst])` to `r.[kf]`.
35. If `relevantExtensionKeys` contains "kn", then
   a. If `SameValue(r.[kn], "true")` is `true` or `r.[kn]` is the empty String, then
      i. Set `locale.([Numeric])` to `true`.
   b. Else,
      i. Set `locale.([Numeric])` to `false`.
36. Set `locale.([NumberingSystem])` to `r.[nu]`.
37. Return `locale`.

### 14.2 Properties of the Intl.Locale Constructor

The Intl.Locale constructor has the following properties:

#### 14.2.1 Intl.Locale.prototype

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

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

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

14.3 Properties of the Intl.Locale Prototype Object

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

14.3.1 Intl.Locale.prototype.constructor

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

14.3.2 Intl.Locale.prototype[@@toStringTag]

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

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

14.3.3 Intl.Locale.prototype.maximize ()

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let maximal be the result of the Add Likely Subtags algorithm applied to loc.[[Locale]]. If an error is signaled, set maximal to loc.[[Locale]].
4. Return ! Construct(%Locale%, maximal).

14.3.4 Intl.Locale.prototype.minimize ()

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let minimal be the result of the Remove Likely Subtags algorithm applied to loc.[[Locale]]. If an error is signaled, set minimal to loc.[[Locale]].
4. Return ! Construct(%Locale%, minimal).

14.3.5 Intl.Locale.prototype.toString ()

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

14.3.6 get Intl.Locale.prototype.baseName

Intl.Locale.prototype.baseName is an accessor property whose set accessor function is undefined. The following algorithm refers to UTS 35’s Unicode Language and Locale Identifiers grammar. Its get accessor function performs the following steps:

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let locale be loc.[[Locale]].
4. Return the substring of  
locale  
corresponding to the  
unicode_language_id  
production.

14.3.7  
get Intl.Locale.prototype.calendar

Intl.Locale.prototype.calendar is an  
accessor property  
whose set accessor function is undefined. Its  
get accessor function performs the following steps:
1. Let  
loc  
be the  
this  
value.
2. Perform  
\textsf{RequireInternalSlot}(loc, [[InitializedLocale]]).
3. Return  
loc.\[[\text{Calendar}]\].

14.3.8  
get Intl.Locale.prototype.caseFirst

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

Intl.Locale.prototype.caseFirst is an  
accessor property  
whose set accessor function is  
\texttt{undefined}. Its  
get accessor function performs the following steps:
1. Let  
loc  
be the  
this  
value.
2. Perform  
\textsf{RequireInternalSlot}(loc, [[InitializedLocale]]).
3. Return  
loc.\[[\text{CaseFirst}]\].

14.3.9  
get Intl.Locale.prototype.collation

Intl.Locale.prototype.collation is an  
accessor property  
whose set accessor function is  
\texttt{undefined}. Its  
get accessor function performs the following steps:
1. Let  
loc  
be the  
this  
value.
2. Perform  
\textsf{RequireInternalSlot}(loc, [[InitializedLocale]]).
3. Return  
loc.\[[\text{Collation}]\].

14.3.10  
get Intl.Locale.prototype.hourCycle

Intl.Locale.prototype.hourCycle is an  
accessor property  
whose set accessor function is  
\texttt{undefined}. Its  
get accessor function performs the following steps:
1. Let  
loc  
be the  
this  
value.
2. Perform  
\textsf{RequireInternalSlot}(loc, [[InitializedLocale]]).
3. Return  
loc.\[[\text{HourCycle}]\].

14.3.11  
get Intl.Locale.prototype.numeric

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

Intl.Locale.prototype.numeric is an  
accessor property  
whose set accessor function is  
\texttt{undefined}. Its  
get accessor function performs the following steps:
1. Let  
loc  
be the  
this  
value.
2. Perform  
\textsf{RequireInternalSlot}(loc, [[InitializedLocale]]).
3. Return  
loc.\[[\text{Numeric}]\].
14.3.12 get Intl.Locale.prototype.numberingSystem

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

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

14.3.13 get Intl.Locale.prototype.language

Intl.Locale.prototype.language is an accessor property whose set accessor function is undefined. The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. Its get accessor function performs the following steps:

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let locale be loc.[[Locale]].
4. Assert: locale matches the unicode_locale_id production.
5. Return the substring of locale corresponding to the unicode_language_subtag production of the unicode_language_id.

14.3.14 get Intl.Locale.prototype.script

Intl.Locale.prototype.script is an accessor property whose set accessor function is undefined. The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. Its get accessor function performs the following steps:

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let locale be loc.[[Locale]].
4. Assert: locale matches the unicode_locale_id production.
5. If the unicode_language_id production of locale does not contain the "-" unicode_script_subtag sequence, return undefined.
6. Return the substring of locale corresponding to the unicode_script_subtag production of the unicode_language_id.

14.3.15 get Intl.Locale.prototype.region

Intl.Locale.prototype.region is an accessor property whose set accessor function is undefined. The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. Its get accessor function performs the following steps:

1. Let loc be the this value.
2. Perform ? RequireInternalSlot(loc, [[InitializedLocale]]).
3. Let locale be loc.[[Locale]].
4. Assert: locale matches the unicode_locale_id production.
5. If the unicode_language_id production of locale does not contain the "-" unicode_region_subtag sequence, return undefined.
6. Return the substring of locale corresponding to the unicode_region_subtag production of the unicode_language_id.
15 NumberFormat Objects

15.1 Abstract Operations for NumberFormat Objects

15.1.1 SetNumberFormatDigitOptions ( \textit{intObj}, \textit{options}, \textit{mnfdDefault}, \textit{mxfdDefault}, \textit{notation} )

The abstract operation \textit{SetNumberFormatDigitOptions} applies digit options used for number formatting onto the \textit{intObj}.

1. \textbf{Assert:} \texttt{Type(intObj)} is Object.
2. \textbf{Assert:} \texttt{Type(options)} is Object.
3. \textbf{Assert:} \texttt{Type(mnfdDefault)} is Number.
4. \textbf{Assert:} \texttt{Type(mxfdDefault)} is Number.
5. Let \textit{mnfd} be \texttt{GetNumberOption(options, "minimumIntegerDigits", 1, 21, 1)}.
6. Let \textit{mnfd} be \texttt{Get(options, "minimumFractionDigits")}.
7. Let \textit{mxfd} be \texttt{Get(options, "maximumFractionDigits")}.
8. Let \textit{mnfd} be \texttt{Get(options, "minimumSignificantDigits")}.
9. Let \textit{mxsd} be \texttt{Get(options, "maximumSignificantDigits")}.
10. Set \texttt{intObj.[[MinimumIntegerDigits]]} to \textit{mnfd}.
11. If \textit{mnfd} is not undefined or \textit{mxsd} is not undefined, then
   a. Set \texttt{intObj.[[RoundingType]]} to significantDigits.
   b. Let \textit{mnfd} be \texttt{DefaultNumberOption(mnfd, 1, 21, 1)}.
   c. Let \textit{mxsd} be \texttt{DefaultNumberOption(mxsd, mnfd, 21, 21)}.
   d. Set \texttt{intObj.[[MinimumSignificantDigits]]} to \textit{mnfd}.
   e. Set \texttt{intObj.[[MaximumSignificantDigits]]} to \textit{mxsd}.
12. Else if \textit{mnfd} is not undefined or \textit{mxfd} is not undefined, then
   a. Set \texttt{intObj.[[RoundingType]]} to fractionDigits.
   b. Let \textit{mnfd} be \texttt{DefaultNumberOption(mnfd, 0, 20, undefined)}.
   c. Let \textit{mxsd} be \texttt{DefaultNumberOption(mxsd, mnfd, 21, 21)}.
   d. If \textit{mnfd} is undefined, set \textit{mnfd} to \texttt{min(mnfdDefault, mxfd)}.
   e. Else if \textit{mxsd} is undefined, set \textit{mxsd} to \texttt{max(mxsdDefault, mnfd)}.
   f. Else if \textit{mnfd} is greater than \textit{mxfd}, throw a \texttt{RangeError} exception.
   g. Set \texttt{intObj.[[MinimumFractionDigits]]} to \textit{mnfd}.
   h. Set \texttt{intObj.[[MaximumFractionDigits]]} to \textit{mxsd}.
13. Else if \textit{notation} is "\texttt{compact}" , then
   a. Set \texttt{intObj.[[RoundingType]]} to compactRounding.
14. Else,
   a. Set \texttt{intObj.[[RoundingType]]} to fractionDigits.
   b. Set \texttt{intObj.[[MinimumFractionDigits]]} to \texttt{mnfdDefault}.
   c. Set \texttt{intObj.[[MaximumFractionDigits]]} to \texttt{mxfdDefault}.

15.1.2 InitializeNumberFormat ( \textit{numberFormat}, \textit{locales}, \textit{options} )

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

The following algorithm refers to the \texttt{type} nonterminal from UTS 35's Unicode Locale Identifier grammar.

1. Let \texttt{requestedLocales} be \texttt{CanonicalizeLocaleList(locales)}.
2. Set \texttt{options} to \texttt{CoerceOptionsToObject(options)}.
3. Let \texttt{opt} be a new Record.
4. Let \texttt{matcher} be \texttt{GetOption(options, "localeMatcher", "string", « "lookupup", "best fit" », "best fit" »)}.
5. Set \texttt{opt.[[localeMatcher]]} to \texttt{matcher}.
6. Let \texttt{numberingSystem} be \texttt{GetOption(options, "numberingSystem", "string", undefined, undefined)}.
7. If `numberingSystem` is not `undefined`, then
   a. If `numberingSystem` does not match the Unicode Locale Identifier type nonterminal, throw a
      `RangeError` exception.
8. Set `opt.[null]` to `numberingSystem`.
9. Let `localeData` be `%NumberFormat%.[[LocaleData]]
10. Let `r` be `ResolveLocale(%NumberFormat%.[[AvailableLocales]], requestedLocales, opt, %NumberFormat%.[[RelevantExtensionKeys]], localeData).
11. Set `numberFormat.[[[Locales]]]` to `r.[[[locale]]]
12. Set `numberFormat.[[[DataLocale]]]` to `r.[[[dataLocale]]]
13. Set `numberFormat.[[[NumberingSystem]]]` to `r.[[[nu]]]
14. Perform ? `SetNumberFormatUnitOptions(numberFormat, options)`.
15. Let `style` be `numberFormat.[[[Style]]]
16. If `style` is "currency", then
   a. Let `currency` be `numberFormat.[[[Currency]]]
   b. Let `cDigits` be `CurrencyDigits(currency)`
   c. Let `mnfdDefault` be `cDigits`
   d. Let `mxfdDefault` be `cDigits`
17. Else,
   a. Let `mnfdDefault` be `0`
   b. If `style` is "percent", then
      i. Let `mxfdDefault` be `0`
   c. Else,  
      i. Let `mxfdDefault` be `3`
19. Set `numberFormat.[[[Notation]]]` to `notation`
20. Perform ? `SetNumberFormatDigitOptions(numberFormat, options, mnfdDefault, mxfdDefault, notation)
22. If `notation` is "compact", then
   a. Set `numberFormat.[[[CompactDisplay]]]` to `compactDisplay`
23. Let `useGrouping` be ? `GetOption(options, "useGrouping", "boolean", undefined, true)
24. Set `numberFormat.[[[UseGrouping]]]` to `useGrouping`
26. Set `numberFormat.[[[SignDisplay]]]` to `signDisplay`
27. Return `numberFormat`

15.1.3 CurrencyDigits ( `currency` )

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

1. If the ISO 4217 currency and funds code list contains `currency` as an alphabetic code, return the minor unit value corresponding to the `currency` from the list; otherwise, return 2.

15.1.4 Number Format Functions

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

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

1. Let `nf` be `F.[[[NumberFormat]]]
2. Assert: Type(`nf`) is Object and `nf` has an [[InitializedNumberFormat]] internal slot.
3. If `value` is not provided, let `value` be `undefined`
4. Let `x` be ? `ToNumeric(value)
5. Return ? `FormatNumeric(nf, x)"
The “length” property of a Number format function is 1.

15.1.5 FormatNumericToString ( intObject, x )

The FormatNumericToString abstract operation is called with arguments intObject (which must be an object with [[RoundingType]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], and [[MaximumFractionDigits]] internal slots), and x (which must be a Number or BigInt value), and returns a Record containing two values: x as a String value with digits formatted according to the five formatting parameters in the field [[FormattedString]], and the final floating decimal value of x after rounding has been performed in the field [[RoundedNumber]].

1. If \( x < 0 \) or \( x \) is -0, let isNegative be true; else let isNegative be false.
2. If isNegative, then
   a. Let \( x \) be -\( x \).
3. If intObject.[[RoundingType]] is significantDigits, then
   a. Let result be ToRawPrecision(\( x \), intObject.[[MinimumSignificantDigits]], intObject.[[MaximumSignificantDigits]])
4. Else if intObject.[[RoundingType]] is fractionDigits, then
   a. Let result be ToRawFixed(\( x \), intObject.[[MinimumFractionDigits]], intObject.[[MaximumFractionDigits]])
5. Else,
   a. Assert: intObject.[[RoundingType]] is compactRounding.
   b. Let result be ToRawPrecision(\( x \), 1, 2).
   c. If result.[[IntegerDigitsCount]] > 1, then
      i. Let result be ToRawFixed(\( x \), 0, 0).
6. Let \( x \) be result.[[RoundedNumber]]
7. Let string be result.[[FormattedString]]
8. Let int be result.[[IntegerDigitsCount]]
9. Let minInteger be intObject.[[MinimumIntegerDigits]]
10. If \( int < \) minInteger, then
    a. Let forwardZeros be the String consisting of minInteger–int occurrences of the character "0".
    b. Set string to the string-concatenation of forwardZeros and string.
11. If isNegative, then
    a. Let \( x \) be -\( x \).
12. Return the Record { [[RoundedNumber]]: \( x \), [[FormattedString]]: string }

15.1.6 PartitionNumberPattern ( numberFormat, x )

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

1. Let exponent be 0.
2. If x is NaN, then
   a. Let n be an implementation- and locale-dependent (ILD) String value indicating the NaN value.
3. Else if x is a non-finite Number, then
   a. Let n be an ILD String value indicating infinity.
4. Else,
   a. If numberFormat.[[Style]] is "percent", let \( x \) be 100 × \( x \).
   b. Let exponent be ComputeExponent(numberFormat, \( x \)).
   c. Let \( x \) be \( x \times 10^{-exponent} \)
   d. Let formatNumberResult be FormatNumericToString(numberFormat, \( x \)).
   e. Let n be formatNumberResult.[[FormattedString]].
   f. Let \( x \) be formatNumberResult.[[RoundedNumber]].
5. Let pattern be GetNumberFormatPattern(numberFormat, \( x \)).
6. Let result be a new empty List.
7. Let patternParts be PartitionPattern(pattern).
8. For each Record \( \{ [[\text{Type}]], [[\text{Value}]] \} \) \( \text{patternPart} \) of \( \text{patternParts} \), do
   a. Let \( p \) be \( \text{patternPart}[[\text{Type}]] \).
   b. If \( p \) is "literal", then
      i. Append a new Record \( \{ [[\text{Type}]], "\text{literal}\"], [[\text{Value}]]: \text{patternPart}[[\text{Value}]] \) as the last element of result.
   c. Else if \( p \) is equal to "number", then
      i. Let notationSubParts be PartitionNotationSubPattern(numberFormat, \( x \), \( n \), exponent).
      ii. Append all elements of \( \text{notationSubParts} \) to result.
   d. Else if \( p \) is equal to "plusSign", then
      i. Let plusSignSymbol be the ILND String representing the plus sign.
      ii. Append a new Record \( \{ [[\text{Type}]], \"\text{plusSign}\"], [[\text{Value}]]: \text{plusSignSymbol} \) as the last element of result.
   e. Else if \( p \) is equal to "minusSign", then
      i. Let minusSignSymbol be the ILND String representing the minus sign.
      ii. Append a new Record \( \{ [[\text{Type}]], \"\text{minusSign}\"], [[\text{Value}]]: \text{minusSignSymbol} \) as the last element of result.
   f. Else if \( p \) is equal to "percentSign" and numberFormat.[[Style]] is "percent", then
      i. Let percentSignSymbol be the ILND String representing the percent sign.
      ii. Append a new Record \( \{ [[\text{Type}]], \"\text{percentSign}\"], [[\text{Value}]]: \text{percentSignSymbol} \) as the last element of result.
   g. Else if \( p \) is equal to "unitPrefix" and numberFormat.[[Style]] is "unit", then
      i. Let unit be numberFormat.[[Unit]].
      ii. Let unitDisplay be numberFormat.[[UnitDisplay]].
      iii. Let \( \mu \) be an ILD String value representing \( \text{unit} \) before \( x \) in unitDisplay form, which may depend on \( x \) in languages having different plural forms.
      iv. Append a new Record \( \{ [[\text{Type}]], \"\text{unit}\"], [[\text{Value}]]: \mu \) as the last element of result.
   h. Else if \( p \) is equal to "unitSuffix" and numberFormat.[[Style]] is "unit", then
      i. Let unit be numberFormat.[[Unit]].
      ii. Let unitDisplay be numberFormat.[[UnitDisplay]].
      iii. Let \( \mu \) be an ILD String value representing \( \text{unit} \) after \( x \) in unitDisplay form, which may depend on \( x \) in languages having different plural forms.
      iv. Append a new Record \( \{ [[\text{Type}]], \"\text{unit}\"], [[\text{Value}]]: \mu \) as the last element of result.
   i. Else if \( p \) is equal to "currencyCode" and numberFormat.[[Style]] is "currency", then
      i. Let currency be numberFormat.[[Currency]].
      ii. Let \( \text{cd} \) be currency.
      iii. Append a new Record \( \{ [[\text{Type}]], \"\text{currency}\"], [[\text{Value}]]: \text{cd} \} \) as the last element of result.
   j. Else if \( p \) is equal to "currencyPrefix" and numberFormat.[[Style]] is "currency", then
      i. Let currency be numberFormat.[[Currency]].
      ii. Let currencyDisplay be numberFormat.[[CurrencyDisplay]].
      iii. Let \( \text{cd} \) be an ILD String value representing \( \text{currency} \) before \( x \) in currencyDisplay form, which may depend on \( x \) in languages having different plural forms.
      iv. Append a new Record \( \{ [[\text{Type}]], \"\text{currency}\"], [[\text{Value}]]: \text{cd} \} \) as the last element of result.
   k. Else if \( p \) is equal to "currencySuffix" and numberFormat.[[Style]] is "currency", then
      i. Let currency be numberFormat.[[Currency]].
      ii. Let currencyDisplay be numberFormat.[[CurrencyDisplay]].
      iii. Let \( \text{cd} \) be an ILD String value representing \( \text{currency} \) 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 a new Record \( \{ [[\text{Type}]], \"\text{currency}\"], [[\text{Value}]]: \text{cd} \} \) as the last element of result.
   l. Else,
      i. Let \( \text{unknown} \) be an ILND String based on \( x \) and \( p \).
      ii. Append a new Record \( \{ [[\text{Type}]], \"\text{unknown}\"], [[\text{Value}]]: unknown \} \) as the last element of result.
9. Return result.
15.1.7  PartitionNotationSubPattern ( numberFormat, x, n, exponent )

The PartitionNotationSubPattern abstract operation is called with arguments numberFormat (which must be an object initialized as a NumberFormat), x (which is a numeric value after rounding is applied), n (which is an intermediate formatted string), and exponent (an integer), and creates the corresponding parts for the number and notation according to the effective locale and the formatting options of numberFormat. The following steps are taken:

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

iv. Else if `p` is equal to "`compactSymbol`", then
   1. Let `compactSymbol` be an ILND 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 a new `Record` `{ [[Type]]: "compact", [[Value]]: `compactSymbol` }` as the last element of `result`.

v. Else if `p` is equal to "`compactName`", then
   1. Let `compactName` be an ILND 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 a new `Record` `{ [[Type]]: "compact", [[Value]]: `compactName` }` as the last element of `result`.

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

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

viii. Else,
    1. Let `unknown` be an ILND String based on `x` and `p`.
    2. Append a new `Record` `{ [[Type]]: "unknown", [[Value]]: `unknown` }` as the last element of `result`.

5. Return `result`.

Table 10 — Numbering systems with simple digit mappings

<table>
<thead>
<tr>
<th>Numbering System</th>
<th>Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>adlm</td>
<td>U+1E950 to U+1E959</td>
</tr>
<tr>
<td>ahom</td>
<td>U+11730 to U+11739</td>
</tr>
<tr>
<td>arab</td>
<td>U+0660 to U+0669</td>
</tr>
<tr>
<td>arabext</td>
<td>U+06F0 to U+06F9</td>
</tr>
<tr>
<td>bali</td>
<td>U+1B50 to U+1B59</td>
</tr>
<tr>
<td>beng</td>
<td>U+09E6 to U+09EF</td>
</tr>
<tr>
<td>bhks</td>
<td>U+11C50 to U+11C59</td>
</tr>
<tr>
<td>brah</td>
<td>U+11066 to U+1106F</td>
</tr>
<tr>
<td>cakm</td>
<td>U+11136 to U+1113F</td>
</tr>
<tr>
<td>Numbering System</td>
<td>Digits</td>
</tr>
<tr>
<td>------------------</td>
<td>--------</td>
</tr>
<tr>
<td>cham</td>
<td>U+AA50 to U+AA59</td>
</tr>
<tr>
<td>deva</td>
<td>U+0966 to U+096F</td>
</tr>
<tr>
<td>diak</td>
<td>U+11950 to U+11959</td>
</tr>
<tr>
<td>fullwide</td>
<td>U+FF10 to U+FF19</td>
</tr>
<tr>
<td>gong</td>
<td>U+11DA0 to U+11DA9</td>
</tr>
<tr>
<td>gonom</td>
<td>U+11D50 to U+11D59</td>
</tr>
<tr>
<td>gujr</td>
<td>U+0AE6 to U+0AEF</td>
</tr>
<tr>
<td>guru</td>
<td>U+0A66 to U+0A6F</td>
</tr>
<tr>
<td>hanidec</td>
<td>U+3007, U+4E00, U+4E8C, U+4E09, U+56DB, U+4E94, U+516D, U+4E03, U+516B, U+4E5D</td>
</tr>
<tr>
<td>hmng</td>
<td>U+16B50 to U+16B59</td>
</tr>
<tr>
<td>hmnp</td>
<td>U+1E140 to U+1E149</td>
</tr>
<tr>
<td>java</td>
<td>U+09D0 to U+09D9</td>
</tr>
<tr>
<td>kali</td>
<td>U+0900 to U+0909</td>
</tr>
<tr>
<td>khmr</td>
<td>U+17E0 to U+17E9</td>
</tr>
<tr>
<td>knda</td>
<td>U+0CE6 to U+0CEF</td>
</tr>
<tr>
<td>lana</td>
<td>U+1A80 to U+1A89</td>
</tr>
<tr>
<td>lanatham</td>
<td>U+1A90 to U+1A99</td>
</tr>
<tr>
<td>laoo</td>
<td>U+0ED0 to U+0ED9</td>
</tr>
<tr>
<td>latn</td>
<td>U+0030 to U+0039</td>
</tr>
<tr>
<td>lepc</td>
<td>U+1C40 to U+1C49</td>
</tr>
<tr>
<td>limb</td>
<td>U+1946 to U+194F</td>
</tr>
<tr>
<td>mathbold</td>
<td>U+1D7CE to U+1D7D7</td>
</tr>
<tr>
<td>mathdbl</td>
<td>U+1D7D8 to U+1D7E1</td>
</tr>
<tr>
<td>mathmono</td>
<td>U+1D7F6 to U+1D7FF</td>
</tr>
<tr>
<td>mathsanb</td>
<td>U+1D7EC to U+1D7F5</td>
</tr>
<tr>
<td>mathsans</td>
<td>U+1D7E2 to U+1D7EB</td>
</tr>
<tr>
<td>mlym</td>
<td>U+0D66 to U+0D6F</td>
</tr>
<tr>
<td>modi</td>
<td>U+11650 to U+11659</td>
</tr>
<tr>
<td>mong</td>
<td>U+1810 to U+1819</td>
</tr>
<tr>
<td>mroo</td>
<td>U+16A60 to U+16A69</td>
</tr>
<tr>
<td>Numbering System</td>
<td>Digits</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>mtei</td>
<td>U+ABF0 to U+ABF9</td>
</tr>
<tr>
<td>mymr</td>
<td>U+1040 to U+1049</td>
</tr>
<tr>
<td>mymrshan</td>
<td>U+1090 to U+1099</td>
</tr>
<tr>
<td>mymrtling</td>
<td>U+A9F0 to U+A9F9</td>
</tr>
<tr>
<td>newa</td>
<td>U+11450 to U+11459</td>
</tr>
<tr>
<td>nkoo</td>
<td>U+07C0 to U+07C9</td>
</tr>
<tr>
<td>olck</td>
<td>U+1C50 to U+1C59</td>
</tr>
<tr>
<td>orya</td>
<td>U+0B66 to U+0B6F</td>
</tr>
<tr>
<td>osma</td>
<td>U+104A0 to U+104A9</td>
</tr>
<tr>
<td>rohg</td>
<td>U+10D30 to U+10D39</td>
</tr>
<tr>
<td>saur</td>
<td>U+A8D0 to U+A8D9</td>
</tr>
<tr>
<td>segment</td>
<td>U+1FBF0 to U+1FBF9</td>
</tr>
<tr>
<td>shrd</td>
<td>U+111D0 to U+111D9</td>
</tr>
<tr>
<td>sind</td>
<td>U+112F0 to U+112F9</td>
</tr>
<tr>
<td>sinh</td>
<td>U+0DE6 to U+0DEF</td>
</tr>
<tr>
<td>sora</td>
<td>U+110F0 to U+110F9</td>
</tr>
<tr>
<td>sund</td>
<td>U+1BB0 to U+1BB9</td>
</tr>
<tr>
<td>takr</td>
<td>U+116C0 to U+116C9</td>
</tr>
<tr>
<td>talu</td>
<td>U+19D0 to U+19D9</td>
</tr>
<tr>
<td>tamldec</td>
<td>U+0BE6 to U+0BEF</td>
</tr>
<tr>
<td>telu</td>
<td>U+0C66 to U+0C6F</td>
</tr>
<tr>
<td>thai</td>
<td>U+0E50 to U+0E59</td>
</tr>
<tr>
<td>tibt</td>
<td>U+0F20 to U+0F29</td>
</tr>
<tr>
<td>tirh</td>
<td>U+114D0 to U+114D9</td>
</tr>
<tr>
<td>vaii</td>
<td>U+A620 to U+A629</td>
</tr>
<tr>
<td>wara</td>
<td>U+118E0 to U+118E9</td>
</tr>
<tr>
<td>wcho</td>
<td>U+1E2F0 to U+1E2F9</td>
</tr>
</tbody>
</table>

NOTE 1  The computations rely on String values and locations within numeric strings that are dependent upon the implementation and the effective locale of `numberFormat("ILD")` or upon the implementation, the effective locale, and the numbering system of `numberFormat("ILND")`. The ILD and ILND Strings mentioned, other than those for currency names, must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.
NOTE 2 It is recommended that implementations use the locale provided by the Common Locale Data Repository (available at http://cldr.unicode.org).

15.1.8 FormatNumeric ( numberFormat, x )

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

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

15.1.9 FormatNumericToParts ( numberFormat, x )

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

1. Let parts be ? PartitionNumberPattern(numberFormat, x).
2. Let result be ArrayCreate(0).
3. Let n be 0.
4. For each Record { [[Type]], [[Value]] } part in 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(n), O).
   e. Increment n by 1.
5. Return result.

15.1.10 ToRawPrecision ( x, minPrecision, maxPrecision )

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

1. Let p be maxPrecision.
2. If x = 0, then
   a. Let m be the String consisting of p occurrences of the character "0".
   b. Let e be 0.
   c. Let xFinal be 0.
3. Else,
   a. Let e be the base 10 logarithm of x rounded down to the nearest integer.
   b. Let n be an integer such that \(10^{p-1} \leq n < 10^{p}\) and for which the exact mathematical value of \(n \times 10^{e-p+1} - x\) is as close to zero as possible. If there is more than one such n, pick the one for which \(n \times 10^{e-p+1}\) is larger.
   c. Let m be the String consisting of the digits of the decimal representation of n (in order, with no leading zeroes).
   d. Let xFinal be \(n \times 10^{e-p+1}\).
4. If e ≥ p−1, then
   a. Let m be the string-concatenation of m and \(e-p+1\) occurrences of the character "0".
   b. Let int be e+1.
5. Else if e ≥ 0, then
a. Let \( m \) be the string-concatenation of the first \( e+1 \) characters of \( m \), the character ".", and the remaining \( p-(e+1) \) characters of \( m \).
b. Let \( int \) be \( e+1 \).
6. Else,
a. Assert: \( e < 0 \).
b. Let \( m \) be the string-concatenation of the String value "0.", \(-(e+1)\) occurrences of the character "0", and \( m \).
c. Let \( int \) be 1.
7. If \( m \) contains the character ".", and \( maxPrecision > minPrecision \), then
a. Let \( cut \) be \( maxPrecision - minPrecision \).
b. Repeat, while \( cut > 0 \) and the last character of \( m \) is "0",
   i. Remove the last character from \( m \).
   ii. Decrease \( cut \) by 1.
c. If the last character of \( m \) is ".", then
   i. Remove the last character from \( m \).
8. Return the Record { [[FormattedString]]: \( m \), [[RoundedNumber]]: \( xFinal \), [[IntegerDigitsCount]]: \( int \) }.

15.1.11 ToRawFixed( \( x \), \( minInteger \), \( minFraction \), \( maxFraction \) )

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

1. Let \( f \) be \( maxFraction \).
2. Let \( n \) be an integer for which the exact mathematical value of \( n / 10^f - x \) is as close to zero as possible. If there are two such \( n \), pick the larger \( n \).
3. Let \( xFinal \) be \( n / 10^f \).
4. If \( n = 0 \), let \( m \) be the String "0". Otherwise, let \( m \) be the String consisting of the digits of the decimal representation of \( n \) (in order, with no leading zeroes).
5. If \( f \neq 0 \), then
   a. Let \( k \) be the number of characters in \( m \).
   b. If \( k \leq f \), then
      i. Let \( z \) be the String value consisting of \( f+1-k \) occurrences of the character "0".
      ii. Let \( m \) be the string-concatenation of \( z \) and \( m \).
      iii. Let \( k \) be \( f+1 \).
   c. Let \( a \) be the first \( k-f \) characters of \( m \), and let \( b \) be the remaining \( f \) characters of \( m \).
   d. Let \( m \) be the string-concatenation of \( a \), ",", and \( b \).
   e. Let \( int \) be the number of characters in \( a \).
6. Else, let \( int \) be the number of characters in \( m \).
7. Let \( cut \) be \( maxFraction - minFraction \).
8. Repeat, while \( cut > 0 \) and the last character of \( m \) is "0",
   a. Remove the last character from \( m \).
   b. Decrease \( cut \) by 1.
9. If the last character of \( m \) is ",", then
   a. Remove the last character from \( m \).
10. Return the Record { [[FormattedString]]: \( m \), [[RoundedNumber]]: \( xFinal \), [[IntegerDigitsCount]]: \( int \) }.

15.1.12 UnwrapNumberFormat( \( nf \) )

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

1. Assert: Type(\( nf \)) is Object.
NORMATIVE OPTIONAL

2. If \( nf \) does not have an [[InitializedNumberFormat]] internal slot and 
   ? OrdinaryHasInstance(%NumberFormat%, \( nf \)) is true, then 
   a. Let \( nf \) be ? Get(\( nf \), %Intl%[[FallbackSymbol]]).

3. Perform ? RequireInternalSlot(\( nf \), [[InitializedNumberFormat]]).
4. Return \( nf \).

NOTE See 8.2 Note 1 for the motivation of the normative optional text.

15.1.13 SetNumberFormatUnitOptions ( \( intlObj, options \) )

The abstract operation SetNumberFormatUnitOptions resolves the user-specified options relating to units onto the intl object.

1. Assert: Type(\( intlObj \)) is Object.
2. Assert: Type(\( options \)) is Object.
3. Let \( style \) be ? GetOption(\( options \), "style", "string", « "decimal", "percent", "currency", "unit" », "decimal").
4. Set \( intlObj.[[Style]] \) to \( style \).
5. Let \( currency \) be ? GetOption(\( options \), "currency", "string", undefined, undefined).
6. If \( currency \) is undefined, then 
   a. If \( style \) is "currency", throw a TypeError exception.
    Else, 
   a. If the result of IsWellFormedCurrencyCode(\( currency \)) is false, throw a RangeError exception.
8. Let \( currencyDisplay \) be ? GetOption(\( options \), "currencyDisplay", "string", « "code", "symbol", "narrowSymbol", "name" », "symbol").
10. Let \( unit \) be ? GetOption(\( options \), "unit", "string", undefined, undefined).
11. If \( unit \) is undefined, then 
    a. If \( style \) is "unit", throw a TypeError exception.
    Else, 
    a. If the result of IsWellFormedUnitIdentifier(\( unit \)) is false, throw a RangeError exception.
13. Let \( unitDisplay \) be ? GetOption(\( options \), "unitDisplay", "string", « "short", "narrow", "long" », "short").
14. If \( style \) is "currency", then 
    a. Let \( currency \) be the result of converting \( currency \) to upper case as specified in 6.1.
    b. Set \( intlObj.[[Currency]] \) to \( currency \).
    c. Set \( intlObj.[[CurrencyDisplay]] \) to \( currencyDisplay \).
    d. Set \( intlObj.[[CurrencySign]] \) to \( currencySign \).
15. If \( style \) is "unit", then 
    a. Set \( intlObj.[[Unit]] \) to \( unit \).
    b. Set \( intlObj.[[UnitDisplay]] \) to \( unitDisplay \).

15.1.14 GetNumberFormatPattern ( \( numberFormat, x \) )

The abstract operation GetNumberFormatPattern considers the resolved unit-related options in the number format object along with the final scaled and rounded number being formatted and returns a pattern, a String value as described in 15.3.3.

1. Let \( localeData \) be %NumberFormat%[[LocaleData]].
2. Let \( dataLocale \) be numberFormat.[[DataLocale]].
3. Let \( dataLocaleData \) be localeData.[[<dataLocale>]].
4. Let \( patterns \) be dataLocaleData.[[patterns]].
5. **Assert**: `patterns` is a Record (see 15.3.3).
6. Let `style` be `numberFormat`.[[Style]].
7. If `style` is "percent", then
   a. Let `patterns` be `patterns`.[[percent]].
8. Else if `style` is "unit", then
   a. Let `unit` be `numberFormat`.[[Unit]].
   b. Let `unitDisplay` be `numberFormat`.[[UnitDisplay]].
   c. Let `patterns` be `patterns`.[[unit]].
   d. If `patterns` doesn't have a field `[[<unit>]]`, then
      i. Let `unit` be "fallback".
   e. Let `patterns` be `patterns`.[[<unit>]].
   f. Let `patterns` be `patterns`.[[<unitDisplay>]].
9. 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. Let `patterns` be `patterns`.[[currency]].
   e. If `patterns` doesn't have a field `[[<currency>]]`, then
      i. Let `currency` be "fallback".
   f. Let `patterns` be `patterns`.[[<currency>]].
   g. Let `patterns` be `patterns`.[[<currencyDisplay>]].
   h. Let `patterns` be `patterns`.[[<currencySign>]].
10. Else,
    a. **Assert**: `style` is "decimal".
    b. Let `patterns` be `patterns`.[[decimal]].
11. Let `signDisplay` be `numberFormat`.[[SignDisplay]].
12. If `signDisplay` is "never", then
    a. Let `pattern` be `patterns`.[[zeroPattern]].
13. Else if `signDisplay` is "auto", then
    a. If `x` is 0 or `x` > 0 or `x` is NaN, then
       i. Let `pattern` be `patterns`.[[zeroPattern]].
    b. Else,
       i. Let `pattern` be `patterns`.[[negativePattern]].
14. Else if `signDisplay` is "always", then
    a. If `x` is 0 or `x` > 0 or `x` is NaN, then
       i. Let `pattern` be `patterns`.[[positivePattern]].
    b. Else,
       i. Let `pattern` be `patterns`.[[negativePattern]].
15. Else,
    a. **Assert**: `signDisplay` is "exceptZero".
    b. If `x` is 0 or `x` < 0 or `x` is NaN, then
       i. Let `pattern` be `patterns`.[[zeroPattern]].
    c. Else if `x` > 0, then
       i. Let `pattern` be `patterns`.[[positivePattern]].
    d. Else,
       i. Let `pattern` be `patterns`.[[negativePattern]].
16. Return `pattern`.

### 15.1.15 GetNotationSubPattern ( `numberFormat`, `exponent` )

The abstract operation `GetNotationSubPattern` considers the resolved notation and `exponent`, and returns a String value for the notation sub pattern as described in 15.3.3.

1. Let `localeData` be `%NumberFormat%`.[[LocaleData]].
2. Let `dataLocale` be `numberFormat`.[[DataLocale]].
3. Let `dataLocaleData` be `localeData`.[[<dataLocale>]].
4. Let $notationSubPatterns$ be $dataLocaleData.[[notationSubPatterns]]$.
5. Assert: $notationSubPatterns$ is a Record (see 15.3.3).
6. Let $notation$ be $numberFormat.[[Notation]]$.
7. If $notation$ is "scientific" or $notation$ is "engineering", then
   a. Return $notationSubPatterns.[[scientific]]$.
8. 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 $compactPatterns.[[<exponent>]]$.
9. Else,
   a. Return "(number)".

15.1.16 ComputeExponent ( $numberFormat$, $x$ )

The abstract operation ComputeExponent computes an exponent (power of ten) by which to scale $x$ according the number formatting settings. It handles cases such as 999 rounding up to 1000, requiring a different exponent.

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

15.1.17 ComputeExponentForMagnitude ( $numberFormat$, $magnitude$ )

The abstract operation ComputeExponentHelper 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).

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 $thousands \times 3$.
5. Else,
   a. Assert: $notation$ is "compact".
   b. Let $exponent$ be an implementation- and locale-dependent (ILD) integer by which to scale a number of the given magnitude in compact notation for the current locale.
   c. Return $exponent$. 
15.2 The Intl.NumberFormat Constructor

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

15.2.1 Intl.NumberFormat ([ locales [ , options ] ])

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

1. If NewTarget is undefined, let newTarget be the active function object, else let newTarget be NewTarget.
2. Let numberFormat be ? OrdinaryCreateFromConstructor(newTarget, "%NumberFormat.prototype%", « [[InitializedNumberFormat]], [[Locale]], [[DataLocale]], [[NumberingSystem]], [[Style]], [[Unit]], [[UnitDisplay]], [[Currency]], [[CurrencyDisplay]], [[CurrencySign]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[RoundingType]], [[Notation]], [[CompactDisplay]], [[UseGrouping]], [[SignDisplay]], [[BoundFormat]] »).
4. Let this be the this value.
5. If NewTarget is undefined and ? OrdinaryHasInstance(%NumberFormat%, this) is true, then
   a. Perform ? DefinePropertyOrThrow(this, %Intl%[[FallbackSymbol]], PropertyDescriptor{ [[Value]]: numberFormat, [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }).
   b. Return this.
6. Return numberFormat.

NOTE See 8.2 1 for the motivation of the normative optional text.

15.3 Properties of the Intl.NumberFormat Constructor

The Intl.NumberFormat constructor has the following properties:

15.3.1 Intl.NumberFormat.prototype

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

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

15.3.2 Intl.NumberFormat.supportedLocalesOf ( locales [ , options ] )

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

1. Let availableLocales be %NumberFormat%, [[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).

The value of the "length" property of the supportedLocalesOf method is 1.
15.3.3 Internal slots

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

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

NOTE 1 Unicode Technical Standard 35 describes two locale extension keys that are relevant to number formatting: "cu" for currency 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.

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 6.5. 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 must not contain any characters 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 http://cldr.unicode.org).

15.4 Properties of the Intl.NumberFormat Prototype Object

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

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

15.4.2 Intl.NumberFormat.prototype[@@toStringTag]

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

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

15.4.3 get Intl.NumberFormat.prototype.format

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

1. Let nf be the this value.
2. If Type(nf) is not Object, throw a TypeError exception.
3. Let nf be ? UnwrapNumberFormat(nf).
4. If nf.[[BoundFormat]] is undefined, then
   a. Let F be a new built-in function object as defined in Number Format Functions (15.1.4).
   b. Set F.[[NumberFormat]] to nf.
   c. Set nf.[[BoundFormat]] to F.
5. Return nf.[[BoundFormat]].

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

15.4.4 Intl.NumberFormat.prototype.formatToParts (value)

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

1. Let nf be the this value.
2. Perform ? RequireInternalSlot(nf, [[InitializedNumberFormat]]).
3. Let x be ? ToNumeric(value).
4. Return ? FormatNumericToParts(nf, x).

15.4.5 Intl.NumberFormat.prototype.resolvedOptions ()

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

1. Let nf be the this value.
2. If Type(nf) is not Object, throw a TypeError exception.
3. Let nf be ? UnwrapNumberFormat(nf).
4. Let options be ? OrdinaryObjectCreate(%Object.prototype%).
5. For each row of Table 11, except the header row, in table order, do
   a. Let p be the Property value of the current row.
   b. Let v be the value of nf's internal slot whose name is the Internal Slot value of the current row.
   c. If v is not undefined, then
      i. Perform ? CreateDataPropertyOrThrow(options, p, v).
5. Return options.
### Table 11 — Resolved Options of NumberFormat Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Locale]]</td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td>[[NumberingSystem]]</td>
<td>&quot;numberingSystem&quot;</td>
</tr>
<tr>
<td>[[Style]]</td>
<td>&quot;style&quot;</td>
</tr>
<tr>
<td>[[Currency]]</td>
<td>&quot;currency&quot;</td>
</tr>
<tr>
<td>[[CurrencyDisplay]]</td>
<td>&quot;currencyDisplay&quot;</td>
</tr>
<tr>
<td>[[CurrencySign]]</td>
<td>&quot;currencySign&quot;</td>
</tr>
<tr>
<td>[[Unit]]</td>
<td>&quot;unit&quot;</td>
</tr>
<tr>
<td>[[UnitDisplay]]</td>
<td>&quot;unitDisplay&quot;</td>
</tr>
<tr>
<td>[[MinimumIntegerDigits]]</td>
<td>&quot;minimumIntegerDigits&quot;</td>
</tr>
<tr>
<td>[[MinimumFractionDigits]]</td>
<td>&quot;minimumFractionDigits&quot;</td>
</tr>
<tr>
<td>[[MaximumFractionDigits]]</td>
<td>&quot;maximumFractionDigits&quot;</td>
</tr>
<tr>
<td>[[MinimumSignificantDigits]</td>
<td>&quot;minimumSignificantDigits&quot;</td>
</tr>
<tr>
<td>[[MaximumSignificantDigits]</td>
<td>&quot;maximumSignificantDigits&quot;</td>
</tr>
<tr>
<td>[[UseGrouping]]</td>
<td>&quot;useGrouping&quot;</td>
</tr>
<tr>
<td>[[Notation]]</td>
<td>&quot;notation&quot;</td>
</tr>
<tr>
<td>[[CompactDisplay]]</td>
<td>&quot;compactDisplay&quot;</td>
</tr>
<tr>
<td>[[SignDisplay]]</td>
<td>&quot;signDisplay&quot;</td>
</tr>
</tbody>
</table>

### 15.5 Properties of Intl.NumberFormat Instances

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

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

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

- **[[Locale]]** is a String value with the language tag of the locale whose localization is used for formatting.
- **[[DataLocale]]** is a String value with the language tag of the nearest locale for which the implementation has data to perform the formatting operation. It will be a parent locale of [[Locale]].
- **[[NumberingSystem]]** is a String value with the “type” given in Unicode Technical Standard 35 for the numbering system used for formatting.
- **[[Style]]** is one of the String values "decimal", "currency", "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, as defined by Unicode Technical Standard #35, Part 2, Section 6. 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 Number value indicating the minimum integer digits to be used. Numbers will be padded with leading zeroes if necessary.
- [[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary. These properties are only used when [[RoundingType]] is fractionDigits.
- [[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] are positive integer Number values 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 significantDigits.
- [[UseGrouping]] is a Boolean value indicating whether a grouping separator should be used.
- [[RoundingType]] is one of the values fractionDigits, significantDigits, or compactRounding, indicating which rounding strategy to use. If fractionDigits, the number is rounded according to [[MinimumFractionDigits]] and [[MaximumFractionDigits]], as described above. If significantDigits, the number is rounded according to [[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] as described above. If compactRounding, the number is rounded to 1 maximum fraction digit if there is 1 digit before the decimal separator, and otherwise round to 0 fraction digits.
- [[Notation]] is one of the String values "standard", "scientific", "engineering", or "compact", specifying whether the number should be displayed without scaling, scaled to the units place with the power of ten in scientific notation, scaled to the nearest thousand with the power of ten in scientific notation, or scaled to the nearest locale-dependent compact decimal notation power of ten with the corresponding compact decimal notation affix.
- [[CompactDisplay]] is one of the String values "short" or "long", specifying whether to display compact notation affixes in short form ("5K") or long form ("5 thousand") if formatting with the "compact" notation. It is only used when [[Notation]] has the value "compact".
- [[SignDisplay]] is one of the String values "auto", "always", "never", or "exceptZero", specifying whether to show the sign on negative numbers only, positive and negative numbers including zero, neither positive nor negative numbers, or positive and negative numbers but not zero. In scientific notation, this slot affects the sign display of the mantissa but not the exponent.

Finally, Intl.NumberFormat instances have a [[BoundFormat]] internal slot that caches the function returned by the format accessor (15.4.3).

## 16 PluralRules Objects

### 16.1 Abstract Operations for PluralRules Objects

#### 16.1.1 InitializePluralRules ( pluralRules, locales, options )

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

1. Let requestedLocales be ? CanonicalizeLocaleList(locales).
2. Set options to ? CoerceOptionsToObject(options).
3. Let opt be a new Record.
5. Set opt.([localeMatcher]) to matcher.
7. Set pluralRules.[[Type]] to t.
8. Perform \( \text{SetNumberFormatDigitOptions}(\text{pluralRules}, \text{options}, +0.\_\_3, \text{"standard"}) \).

9. Let \( \text{localeData} \) be %PluralRules%[[LocaleData]].

10. Let \( r \) be ResolveLocale(%PluralRules%[[AvailableLocales]], \text{requestedLocales}, \text{opt}, %PluralRules%[[AvailableLocales]], \text{localeData}).

11. Set \( \text{pluralRules}[[\text{Locale}]] \) to the value of \( r[[\text{locale}]] \).

12. Return \( \text{pluralRules} \).

16.1.2 GetOperands (\( s \))

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

1. \textbf{Assert:} \( \text{Type}(s) \) is String.
2. Let \( n \) be \( \text{ToNumber}(s) \).
3. \textbf{Assert:} \( n \) is finite.
4. Let \( dp \) be \( \text{StringIndexOf}(s, \text{"."}, 0) \).
5. If \( dp = -1 \), then
   a. Set \( iv \) to \( n \).
   b. Let \( f \) be 0.
   c. Let \( v \) be 0.
6. Else,
   a. Let \( iv \) be the substring of \( s \) from position 0, inclusive, to position \( dp \), exclusive.
   b. Let \( fv \) be the substring of \( s \) from position \( dp \), exclusive, to the end of \( s \).
   c. Let \( f \) be \( \text{ToNumber}(fv) \).
   d. Let \( v \) be the length of \( fv \).
7. Let \( i \) be \( \text{abs}(\text{ToNumber}(iv)) \).
8. If \( f \neq 0 \), then
   a. Let \( ft \) be the value of \( fv \) stripped of trailing "0".
   b. Let \( w \) be the length of \( ft \).
   c. Let \( t \) be \( \text{ToNumber}(ft) \).
9. Else,
   a. Let \( w \) be 0.
   b. Let \( t \) be 0.
10. Return a new \textbf{Record} { [[Number]]: \( n \), [[IntegerDigits]]: \( i \), [[NumberOfFractionDigits]]: \( v \), [[NumberOfFractionDigitsWithoutTrailing]]: \( w \), [[FractionDigits]]: \( t \), [[FractionDigitsWithoutTrailing]]: \( t \) }. 
Table 12 — Plural Rules Operands Record Fields

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Number]]</td>
<td>Number</td>
<td>Absolute value of the source number (integer and decimals)</td>
</tr>
<tr>
<td>[[IntegerDigits]]</td>
<td>Number</td>
<td>Number of digits of [[Number]].</td>
</tr>
<tr>
<td>[[NumberOfFractionDigits]]</td>
<td>Number</td>
<td>Number of visible fraction digits in [[Number]], with trailing zeroes.</td>
</tr>
<tr>
<td>[[NumberOfFractionDigitsWithoutTrailing]]</td>
<td>Number</td>
<td>Number of visible fraction digits in [[Number]], without trailing zeroes.</td>
</tr>
<tr>
<td>[[FractionDigits]]</td>
<td>Number</td>
<td>Number of visible fractional digits in [[Number]], with trailing zeroes.</td>
</tr>
<tr>
<td>[[FractionDigitsWithoutTrailing]]</td>
<td>Number</td>
<td>Number of visible fractional digits in [[Number]], without trailing zeroes.</td>
</tr>
</tbody>
</table>

16.1.3 PluralRuleSelect (locale, type, n, operands)

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

16.1.4 ResolvePlural (pluralRules, n)

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

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

16.2 The Intl.PluralRules Constructor

The PluralRules constructor is the %PluralRules% intrinsic object and a standard built-in property of the Intl object. Behaviour common to all service constructor properties of the Intl object is specified in 9.1.
16.2.1 Intl.PluralRules ( [ locales [ , options ] ])

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

1. If NewTarget is undefined, throw a TypeError exception.
2. Let pluralRules be ? OrdinaryCreateFromConstructor(NewTarget, "%PluralRules.prototype%", « [[InitializedPluralRules]], [[Locale]], [[Type]], [[MinimumIntegerDigits]], [[MinimumFractionDigits]], [[MaximumFractionDigits]], [[MinimumSignificantDigits]], [[MaximumSignificantDigits]], [[RoundingType]] »).

16.3 Properties of the Intl.PluralRules Constructor

16.3.1 Intl.PluralRules.prototype

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

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

16.3.2 Intl.PluralRules.supportedLocalesOf ( locales [ , options ])

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

1. Let availableLocales be %PluralRules%.[[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).

The value of the "length" property of the supportedLocalesOf method is 1.

16.3.3 Internal slots

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

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

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

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

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


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

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

16.4.2 Intl.PluralRules.prototype @@toStringTag

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

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

16.4.3 Intl.PluralRules.prototype.select (value)

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

1. Let pr be the this value.
2. Perform ? RequireInternalSlot(pr, [[InitializedPluralRules]]).
3. Let n be ? ToNumber(value).

16.4.4 Intl.PluralRules.prototype.resolvedOptions ()

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

1. Let pr be the this value.
2. Perform ? RequireInternalSlot(pr, [[InitializedPluralRules]]).
3. Let options be ! OrdinaryObjectCreate(%Object.prototype%).
4. For each row of Table 13, except the header row, in table order, do
   a. Let p be the Property value of the current row.
   b. Let v be the value of pr's internal slot whose name is the Internal Slot value of the current row.
   c. If v is not undefined, then
       i. Perform ! CreateDataPropertyOrThrow(options, p, v).
5. Let pluralCategories be a List of Strings representing the possible results of PluralRuleSelect for the selected locale pr.[[Locale]]. This List consists of unique String values, from the list "zero", "one", "two", "few", "many" and "other", that are relevant for the locale whose localization is specified in LDML Language Plural Rules.
6. Perform ! CreateDataProperty(options, "pluralCategories", CreateArrayFromList(pluralCategories)).
7. Return options.

Table 13 — Resolved Options of PluralRules Instances

<table>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[Locale]]</td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td>[[Type]]</td>
<td>&quot;type&quot;</td>
</tr>
<tr>
<td>[[MinimumIntegerDigits]]</td>
<td>&quot;minimumIntegerDigits&quot;</td>
</tr>
<tr>
<td>[[MinimumFractionDigits]]</td>
<td>&quot;minimumFractionDigits&quot;</td>
</tr>
<tr>
<td>[[MaximumFractionDigits]]</td>
<td>&quot;maximumFractionDigits&quot;</td>
</tr>
<tr>
<td>[[MinimumSignificantDigits]]</td>
<td>&quot;minimumSignificantDigits&quot;</td>
</tr>
<tr>
<td>[[MaximumSignificantDigits]]</td>
<td>&quot;maximumSignificantDigits&quot;</td>
</tr>
</tbody>
</table>
16.5 Properties of Intl.PluralRules Instances

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

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

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

- [[Locale]] is a String value with the language tag of the locale whose localization is used by the plural rules.
- [[Type]] is one of the String values "cardinal" or "ordinal", identifying the plural rules used.
- [[MinimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used.
- [[MinimumFractionDigits]] and [[MaximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.
- [[MinimumSignificantDigits]] and [[MaximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be used. Either none or both of these properties are present; if they are, they override minimum and maximum integer and fraction digits.
- [[RoundingType]] is one of the values fractionDigits or significantDigits, indicating which rounding strategy to use, as discussed in 15.5.

17 RelativeTimeFormat Objects

17.1 Abstract Operations for RelativeTimeFormat Objects

17.1.1 InitializeRelativeTimeFormat (relativeTimeFormat, locales, options)

The abstract operation InitializeRelativeTimeFormat accepts the arguments relativeTimeFormat (which must be an object), locales, and options. It initializes relativeTimeFormat as a RelativeTimeFormat object.

The following algorithm refers to UTS 35's Unicode Language and Locale Identifiers grammar. The following steps are taken:

1. Let requestedLocales be ? CanonicalizeLocaleList(locales).
2. Set options to ? CoerceOptionsToObject(options).
3. Set opt to a new Record.
5. Set opt.[[LocaleMatcher]] to matcher.
7. If numberingSystem is not undefined, then
   a. If numberingSystem does not match the type sequence (from UTS 35 Unicode Locale Identifier, section 3.2), throw a RangeError exception.
8. Set opt.[[nu]] to numberingSystem.
9. Let localeData be %RelativeTimeFormat%.[[LocaleData]].
10. Let r be ResolveLocale(%RelativeTimeFormat%, [[AvailableLocales]], requestedLocales, opt, %RelativeTimeFormat%.[[RelevantExtensionKeys]], localeData).
11. Let locale be r.[[locale]].
12. Set relativeTimeFormat.[[Locale]] to locale.
13. Set relativeTimeFormat.[[DataLocale]] to r.[[dataLocale]].
14. Set relativeTimeFormat.[[NumberingSystem]] to r.[[nu]].
15. Let style be ? GetOption(options, "style", "string", "long", "short", "narrow", "long").
16. Set relativeTimeFormat.[[Style]] to style.
17. Let numeric be ? GetOption(options, "numeric", "string", «"always", "auto"», "always").
18. Set relativeTimeFormat.[[Numeric]] to numeric.
19. Let relativeTimeFormat.[[NumberFormat]] be ! Construct(%NumberFormat%, «locale»).
20. Let relativeTimeFormat.[[PluralRules]] be ! Construct(%PluralRules%, «locale»).

17.1.2 SingularRelativeTimeUnit (unit)

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

17.1.3 PartitionRelativeTimePattern (relativeTimeFormat, value, unit)

When the PartitionRelativeTimePattern abstract operation is called with arguments relativeTimeFormat, value, and unit it returns a String value representing value (which must be a Number value) according to the effective locale and the formatting options of relativeTimeFormat.

1. Assert: relativeTimeFormat has an [[InitializedRelativeTimeFormat]] internal slot.
2. Assert: Type(value) is Number.
3. Assert: Type(unit) is String.
4. If value is NaN, +∞, or -∞, throw a RangeError exception.
5. Let unit be ? SingularRelativeTimeUnit(unit).
6. Let localeData be %RelativeTimeFormat%, [[LocaleData]].
7. Let dataLocale be relativeTimeFormat.[[DataLocale]].
8. Let fields be localeData.[[<dataLocale>]].
9. Let style be relativeTimeFormat.[[Style]].
10. If style is equal to "short", then
    a. Let entry be the string-concatenation of unit and "-short".
11. Else if style is equal to "narrow", then
    a. Let entry be the string-concatenation of unit and "-narrow".
12. Else,
    a. Let entry be unit.
13. If fields doesn't have a field [[<entry>]], then
    a. Let entry be unit.
14. Let patterns be fields.[[<entry>]].
15. Let numeric be relativeTimeFormat.[[Numeric]].
16. If numeric is equal to "auto", then
    a. Let valueString be ToString(value).
    b. If patterns has a field [[<valueString>]], then
        i. Let result be patterns.[[<valueStrings>]].
        ii. Return a List containing the Record { [[Type]]: "literal", [[Value]]: result }.
17. If value is +0F or if value is less than 0, then
    a. Let // be "past".
18. Else,
    a. Let // be "future".
Let \( po \) be patterns.\([\textbf{<tl>}]\).

Let \( fv \) be \( \text{PartitionNumberPattern}(\text{relativeTimeFormat.\([\textbf{NumberFormat}]\), value}) \).

Let \( pr \) be \( \text{ResolvePlural}(\text{relativeTimeFormat.\([\textbf{PluralRules}]\), value}) \).

Let \( pattern \) be \( po.\([\textbf{<pr>}]\) \).

Return \( \text{MakePartsList}(pattern, unit, fv) \).

17.1.4 \textit{MakePartsList ( pattern, unit, parts )}

The MakePartsList abstract operation is called with arguments \( pattern \), a pattern String, \( unit \), a String, and \( parts \), a List of Records representing a formatted Number.

Example:

\( \text{MakePartsList("AA(0)BB", "hour", « \{ \textbf{[Type]}: "integer", \textbf{[Value]}: "15" \} » \) } \)

Output (List of Records):

«

\( \{ \textbf{[Type]}: "literal", \textbf{[Value]}: "AA", \textbf{[Unit]}: empty \}\),

\( \{ \textbf{[Type]}: "integer", \textbf{[Value]}: "15", \textbf{[Unit]}: "hour" \}\),

\( \{ \textbf{[Type]}: "literal", \textbf{[Value]}: "BB", \textbf{[Unit]}: empty \}\)

»

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

17.1.5 \textit{FormatRelativeTime ( relativeTimeFormat, value, unit )}

The FormatRelativeTime abstract operation is called with arguments \( relativeTimeFormat \) (which must be an object initialized as a RelativeTimeFormat), \( value \) (which must be a Number value), and \( unit \) (which must be a String denoting the value unit) and performs the following steps:

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

The FormatRelativeTimeToParts abstract operation is called with arguments relativeTimeFormat (which must be an object initialized as a RelativeTimeFormat), value (which must be a Number value), and unit (which must be a String denoting the value unit) and performs the following steps:

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 in parts, do
   a. Let O be OrdinaryObjectCreate(%Object.prototype%).
   b. Perform ! CreateDataPropertyOrThrow(O, "type", part.[[Type]]).
   c. Perform ! CreateDataPropertyOrThrow(O, "value", part.[[Value]]).
   d. If part.[[Unit]] is not empty, then
      i. Perform ! CreateDataPropertyOrThrow(O, "unit", part.[[Unit]]).
   e. Perform ! CreateDataPropertyOrThrow(result, ! ToString(n), O).
   f. Increment n by 1.
5. Return result.

17.2 The Intl.RelativeTimeFormat Constructor

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

17.2.1 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, "%RelativeTimeFormat.prototype%", « [[InitializedRelativeTimeFormat]], [[Locale]], [[DataLocale]], [[Style]], [[Numeric]], [[NumberFormat]], [[NumberingSystem]], [[PluralRules]] »).

17.3 Properties of the Intl.RelativeTimeFormat Constructor

The Intl.RelativeTimeFormat constructor has the following properties:

17.3.1 Intl.RelativeTimeFormat.prototype

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

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

17.3.2 Intl.RelativeTimeFormat.supportedLocalesOf([ locales [ , options ] ])

When the supportedLocalesOf method of %RelativeTimeFormat% is called, the following steps are taken:

1. Let availableLocales be %RelativeTimeFormat%.[[AvailableLocales]].
2. Let requestedLocales be ? CanonicalizeLocaleList(locales).
The value of the "length" property of the supportedLocalesOf method is 1.

17.3.3  Internal slots

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

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

NOTE 1  Unicode Technical Standard 35 describes 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]] 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", "tradition", or "finance".

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

17.4  Properties of the Intl.RelativeTimeFormat PrototypeObject

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

17.4.1  Intl.RelativeTimeFormat.prototype.constructor

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

17.4.2  Intl.RelativeTimeFormat.prototype[ @@toStringTag ]

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

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

17.4.3  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).
Return `FormatRelativeTime(relativeTimeFormat, value, unit)`. 

### 17.4.4 Intl.RelativeTimeFormat.prototype.formatToParts (value, unit)

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

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

### 17.4.5 Intl.RelativeTimeFormat.prototype.resolvedOptions()

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

1. Let `relativeTimeFormat` be the this value.
2. Perform `RequireInternalSlot(relativeTimeFormat, [[InitializedRelativeTimeFormat]])`.
3. Let `options` be `! OrdinaryObjectCreate(%Object.prototype%)`.
4. For each row of Table 14, 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>
<thead>
<tr>
<th>Internal Slot</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[[Locale]]</code></td>
<td>&quot;locale&quot;</td>
</tr>
<tr>
<td><code>[[Style]]</code></td>
<td>&quot;style&quot;</td>
</tr>
<tr>
<td><code>[[Numeric]]</code></td>
<td>&quot;numeric&quot;</td>
</tr>
<tr>
<td><code>[[NumberingSystem]]</code></td>
<td>&quot;numberingSystem&quot;</td>
</tr>
</tbody>
</table>

### 17.5 Properties of Intl.RelativeTimeFormat Instances

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

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

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

- `[[Locale]]` is a String value with the language tag of the locale whose localization is used for formatting.
- `[[DataLocale]]` is a String value with the language tag of the nearest locale for which the implementation has data to perform the formatting operation. It will be a parent locale of `[[Locale]]`.
- `[[Style]]` is one of the String values "long", "short", or "narrow", identifying the relative time format style used.
18  Locale Sensitive Functions of the ECMAScript Language Specification

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

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

18.1  Properties of the String Prototype Object

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

This definition supersedes the definition provided in ES2021, 21.1.3.10.

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

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

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

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

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

18.1.2  String.prototype.toLocaleLowerCase ( [ locales ])

This definition supersedes the definition provided in ES2021, 21.1.3.23.

This function interprets a String value as a sequence of code points, as described in ES2021, 6.1.4. The following steps are taken:

1. Let `O` be RequireObjectCoercible(this value).
2. Let `S` be ? ToString(O).
4. If `requestedLocales` is not an empty List, then
   a. Let `requestedLocale` be requestedLocales[0].
5. Else,
   a. Let `requestedLocale` be DefaultLocale().

• [[Numeric]] is one of the String values "always" or "auto", identifying whether numerical descriptions are always used, or used only when no more specific version is available (e.g., "1 day ago" vs "yesterday").
• [[NumberFormat]] is an Intl.NumberFormat object used for formatting.
• [[NumberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
• [[PluralRules]] is an Intl.PluralRules object used for formatting.

18  Locale Sensitive Functions of the ECMAScript Language Specification

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

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

18.1  Properties of the String Prototype Object

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

This definition supersedes the definition provided in ES2021, 21.1.3.10.

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

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

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

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

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

18.1.2  String.prototype.toLocaleLowerCase ( [ locales ])

This definition supersedes the definition provided in ES2021, 21.1.3.23.

This function interprets a String value as a sequence of code points, as described in ES2021, 6.1.4. The following steps are taken:

1. Let `O` be RequireObjectCoercible(this value).
2. Let `S` be ? ToString(O).
4. If `requestedLocales` is not an empty List, then
   a. Let `requestedLocale` be requestedLocales[0].
5. Else,
   a. Let `requestedLocale` be DefaultLocale().
6. Let $noExtensionsLocale$ be the String value that is $requestedLocale$ with all Unicode locale extension sequences (6.2.1) removed.
7. Let $availableLocales$ be a List with language tags that includes the languages for which the Unicode Character Database contains language sensitive case mappings. Implementations may add additional language tags if they support case mapping for additional locales.
8. Let $locale$ be $BestAvailableLocale(availableLocales, noExtensionsLocale)$.
9. If $locale$ is undefined, let $locale$ be "und".
10. Let $cpList$ be a List containing in order the code points of $S$ as defined in ES2021, 6.1.4, starting at the first element of $S$.
11. Let $cuList$ be a List where the elements are the result of a lower case transformation of the ordered code points in $cpList$ according to the Unicode Default Case Conversion algorithm or an implementation-defined conversion algorithm. A conforming implementation's lower case transformation algorithm must always yield the same $cpList$ given the same $cuList$ and $locale$.
12. Let $L$ be a String whose elements are the UTF-16 Encoding (defined in ES2021, 6.1.4) of the code points of $cuList$.
13. Return $L$.

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

NOTE 1 The case mapping of some code points may produce multiple code points. In this case the result String may not be the same length as the source String. Because both $toLocaleUpperCase$ and $toLocaleLowerCase$ have context-sensitive behaviour, the functions are not symmetrical. In other words, $s.toLocaleUpperCase()$ . $toLocaleLowerCase()$ is not necessarily equal to $s.toLocaleLowerCase()$.

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

### 18.1.3 String.prototype.toLocaleUpperCase ([ locales ])

This definition supersedes the definition provided in ES2021, 21.1.3.24.

This function interprets a String value as a sequence of code points, as described in ES2021, 6.1.4. This function behaves in exactly the same way as $String.prototype.toLocaleLowerCase$, except that characters are mapped to their $toUpperCase$ equivalents. A conforming implementation's upper case transformation algorithm must always yield the same result given the same sequence of code points and $locale$.

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

### 18.2 Properties of the Number Prototype Object

The following definition(s) refer to the abstract operation $thisNumberValue$ as defined in ES2021, 20.1.3.

#### 18.2.1 Number.prototype.toLocaleString ([ locales [, options ] ])

This definition supersedes the definition provided in ES2021, 20.1.3.4.

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

1. Let $x$ be $thisNumberValue(this value).
2. Let $numberFormat$ be $Construct(%NumberFormat%, « locales, options »)$.
3. Return $FormatNumeric(numberFormat, x)$.
18.3 Properties of the BigInt Prototype Object

The following definition(s) refer to the abstract operation \texttt{thisBigIntValue} as defined in ES2021, 20.2.3.

18.3.1 BigInt.prototype.toLocaleString ([ \texttt{locales} [, \texttt{options}] ]) 

This definition supersedes the definition provided in ES2021, 20.2.3.2.

When the \texttt{toLocaleString} method is called with optional arguments \texttt{locales} and \texttt{options}, the following steps are taken:

1. Let \( x \) be \( \texttt{thisBigIntValue(this value)} \).
2. Let \( \texttt{numberFormat} \) be \( \texttt{Construct}%\texttt{NumberFormat}, \texttt{« locales, options »} \).
3. Return \( \texttt{FormatNumeric(numberFormat, x)} \).

18.4 Properties of the Date Prototype Object

The following definition(s) refer to the abstract operation \texttt{thisTimeValue} as defined in ES2021, 20.4.4.

18.4.1 Date.prototype.toLocaleString ([ \texttt{locales} [, \texttt{options}] ]) 

This definition supersedes the definition provided in ES2021, 20.4.4.39.

When the \texttt{toLocaleString} method is called with optional arguments \texttt{locales} and \texttt{options}, the following steps are taken:

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

18.4.2 Date.prototype.toLocaleDateString ([ \texttt{locales} [, \texttt{options}] ]) 

This definition supersedes the definition provided in ES2021, 20.4.4.38.

When the \texttt{toLocaleDateString} method is called with optional arguments \texttt{locales} and \texttt{options}, the following steps are taken:

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

18.4.3 Date.prototype.toLocaleTimeString ([ \texttt{locales} [, \texttt{options}] ]) 

This definition supersedes the definition provided in ES2021, 20.4.4.40.

When the \texttt{toLocaleTimeString} method is called with optional arguments \texttt{locales} and \texttt{options}, the following steps are taken:

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

18.5 Properties of the Array Prototype Object

18.5.1 \( \text{Array.prototype.toLocaleString} ([ locales [, options ] ] \))

This definition supersedes the definition provided in ES2021, 22.1.3.29.

When the \( \text{toLocaleString} \) method is called with optional arguments \( locales \) and \( options \), the following steps are taken:

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

NOTE 1 This algorithm's steps mirror the steps taken in 22.1.3.29, with the exception that \( \text{Invoke}(nextElement, "toLocaleString") \) now takes \( locales \) and \( options \) as arguments.

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

NOTE 3 The \( \text{toLocaleString} \) function is intentionally generic; it does not require that its \( \text{this} \) value be an Array object. Therefore it can be transferred to other kinds of objects for use as a method.
Annex A

Implementation Dependent Behaviour

The following aspects of the ECMAScript 2021 Internationalization API Specification are implementation dependent:

- In all functionality:
  - Additional values for some properties of `options` arguments (2)
  - The default locale (6.2.4)
  - The default time zone (6.4.3)
  - The set of available locales for each constructor (9.1)
  - The `BestFitMatcher` algorithm (9.2.4)
  - The `BestFitSupportedLocales` algorithm (9.2.9)

- 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 sort order for each supported locale and options combination (10.3.3.1)

- In DateTimeFormat:
  - The `BestFitFormatMatcher` algorithm (11.1.1)
  - The set of supported "ca" key values (calendars) per locale (11.3.3)
  - The set of supported "nu" key values (numbering systems) per locale (11.3.3)
  - The default hourCycle setting per locale (11.3.3)
  - The set of supported date-time formats per locale beyond a core set, including the representations used for each component and the associated patterns (11.3.3)
  - Localized weekday names, era names, month names, day period names, am/pm indicators, and time zone names (11.1.9)
  - The calendric calculations used for calendars other than "gregory", and adjustments for local time zones and daylight saving time (11.1.9)

- In DisplayNames:
  - The localized names (12.3.3)

- In ListFormat:
  - The patterns used for formatting values (13.3.3)

- In Locale:
  - Support for the Unicode extensions keys "kf", "kn" and the parallel options properties "caseFirst", "numeric" (14.1.3)

- In NumberFormat:
  - The set of supported "nu" key values (numbering systems) per locale (15.3.3)
  - The patterns used for formatting values as decimal, percent, currency, or unit values per locale, with or without the sign, with or without accounting format for currencies, and in standard, compact, or scientific notation (15.1.8)
  - Localized representations of `NaN` and `Infinity` (15.1.8)
  - The implementation of numbering systems not listed in Table 10 (15.1.8)
  - Localized decimal and grouping separators (15.1.8)
  - Localized plus and minus signs (15.1.8)
- Localized digit grouping schemata (15.1.8)
- Localized magnitude thresholds for compact notation (15.1.8)
- Localized symbols for compact and scientific notation (15.1.8)
- Localized narrow, short, and long currency symbols and names (15.1.8)
- Localized narrow, short, and long unit symbols (15.1.8)

- In PluralRules:
  - List of Strings representing the possible results of plural selection and their corresponding order per locale. (16.1.1)

- In RelativeTimeFormat:
  - The set of supported "nu" key values (numbering systems) per locale (17.3.3)
  - The patterns used for formatting values (17.3.3)
Annex B

Additions and Changes That Introduce Incompatibilities with Prior Editions

- **10.1, 15.2, 11.2** In ECMA-402, 1st Edition, constructors could be used to create Intl objects from arbitrary objects. This is no longer possible in 2nd Edition.
- **11.4.3** 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 $1_F$.
- **10.3.2** In ECMA-402, 7th Edition, the `@@toStringTag` property of Intl.Collator.prototype was set to "Object". In 8th Edition, `@@toStringTag` is set to "Intl.Collator".
- **11.4.2** In ECMA-402, 7th Edition, the `@@toStringTag` property of Intl.DateTimeFormat.prototype was set to "Object". In 8th Edition, `@@toStringTag` is set to "Intl.DateTimeFormat".
- **15.4.2** In ECMA-402, 7th Edition, the `@@toStringTag` property of Intl.NumberFormat.prototype was set to "Object". In 8th Edition, `@@toStringTag` is set to "Intl.NumberFormat".
- **16.4.2** In ECMA-402, 7th Edition, the `@@toStringTag` property of Intl.PluralRules.prototype was set to "Object". In 8th Edition, `@@toStringTag` is set to "Intl.PluralRules".
- **8.1.1** In ECMA-402, 7th Edition, the `@@toStringTag` property of Intl was not defined. In 8th Edition, `@@toStringTag` is set to "Intl".
- **15.2** 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

Colophon

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

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

Copyright & Software License

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

Copyright Notice

© 2021 Ecma International

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.

© Ecma International 2021
Software License

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.

1. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
2. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
3. 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.

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.