ecma INTERNATIONAL

Ecma/TC39/2011/NN

Working Draft Standard ECMA-XXX

1st Edition / Draft 1 December 2011

ECMAScript Globalization API Specification

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

Contents

Page

"DISCLAIMER

This draft document may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to Ecma International, except as needed for the purpose of developing any document or deliverable produced by Ecma International.

This disclaimer is valid only prior to final version of this document. After approval all rights on the standard are reserved by Ecma International.

The limited permissions are granted through the standardization phase and will not be revoked by Ecma International or its successors or assigns during this time.

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

ECMAScript Globalization API Specification

1 Scope

This Standard defines the ECMAScript Globalization API.

2 Conformance

A conforming implementation of the ECMAScript Globalization API must conform to the ECMAScript Language Specification, 5.1 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 Globalization API is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of the ECMAScript Globalization API 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.

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.

ECMA-262, ECMAScript Language Specification, 5.1 edition or successor

ISO/IEC 10646:2003: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005, plus additional amendments and corrigenda, or successor

The Unicode Standard, Version 4.1.0, or successor

IETF BCP 47:
- RFC 5646, Tags for Identifying Languages, or successor
- RFC 4647, Matching of Language Tags, or successor
- IANA Language Subtag Registry, file date 2011-08-25 or later

IETF RFC 6067, BCP 47 Extension U, or successor

Unicode Technical Standard 35, Unicode Locale Data Markup Language, version 2.0.1 or successor

ISO 4217:2008, Codes for the representation of currencies and funds, or successor

4 Overview

This section contains a non-normative overview of the ECMAScript Globalization API.

4.1 Globalization

In software development, globalization is commonly understood to be the combination of internationalization and localization. Internationalization of software means designing it such that it supports or can be easily adapted to support the needs of users speaking different languages and having different cultural expectations, and enables worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization 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 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 behavior of these functions. The ECMAScript Globalization API builds on this by providing an initial 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 Globalization API is designed to complement the ECMAScript Language Specification by providing key language-sensitive functionality. The API can be added to an implementation of the ECMAScript Language Specification, 5.1 edition or successor.

This initial version of the ECMAScript Globalization API provides three key pieces of language-sensitive functionality that are required in most applications: String comparison (collation), number formatting, and date and time formatting. While the ECMAScript Language Specification provides functions for this functionality (String.prototype.localeCompare, Number.prototype.toLocaleString, Date.prototype.toLocaleString, Date.prototype.toLocaleDateString, and Date.prototype.toLocaleTimeString), the Globalization API Specification gives applications control over the language and over details of the behavior to be used.

Applications can use the API in two ways:

1. Directly, by using the constructors Collator, NumberFormat, or DateTimeFormat to construct an object, specifying a list of preferred languages and options to configure the behavior of the resulting object. The object then provides a main function (compare 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 Language Specification mentioned above, which 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.

To support the handling of BCP 47 language tags, LocaleList objects assist with validation and canonicalization of these tags and negotiation against the available locales in an implementation.

The Globalization object is used to package all functionality defined in the ECMAScript Globalization API to avoid name collisions.

4.3 Implementation Dependencies

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

- *The set of locales that are supported 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 globalization 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 behavior 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 behavior.

5 Notational Conventions

This standard uses a subset of the notational conventions of the ECMAScript Language Specification, 5.1 edition:

- Algorithm conventions, including the use of abstract operations, as described in the ECMAScript Language Specification, 5.1 edition, 5.2.

- Internal properties, as described in the ECMAScript Language Specification, 5.1 edition, 8.6.2.

NOTE As described in the ECMAScript Language Specification, algorithms are used to precisely specify the required semantics of ECMAScript constructs, but are not intended to imply the use of any specific implementation technique. Internal properties 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 properties in the manner described here.

6 Identification of Locales, Time Zones, and Currencies

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

6.1 Case Sensitivity and Case Mapping

Implementations shall interpret the String values described in this clause as case-insensitive, 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). Localized case folding, which might introduce other equivalences, shall not be used. When mapping to lower case or upper case, a mapping shall be used that maps characters in the range "A" to "Z" (U+0041 to U+005A) to the corresponding characters in the range "a" to "z" (U+0061 to U+007A), or vice versa.

6.2 Language Tags

The ECMAScript Globalization API identifies locales using language tags as defined by IETF BCP 47 (RFCs 5646 and 4647 or their successors), which may include extensions such as those registered through RFC 6067. Their canonical form is specified in RFC 5646 section 4.5 or its successor.

Implementations shall accept all well-formed BCP 47 language tags, as specified in RFC 5646 section 2.2.9 or successor. However, the set of locales and thus language tags that an implementation supports with adequate localizations is implementation dependent.

6.2.1 IsWellFormedLanguageTag(locale)

The IsWellFormedLanguageTag abstract operation verifies that the locale argument (which it expects to be a String value) represents a well-formed BCP 47 language tag as specified in RFC 5646 section 2.1, or successor. It returns true if locale can be generated from the ABNF grammar in that section, starting with Language-Tag, false otherwise. Terminal value characters in the grammar are interpreted as the Unicode equivalents of the ASCII octet values given.

6.2.2 CanonicalizeLanguageTag (locale)

The CanonicalizeLanguageTag abstract operation returns the canonical and case-regularized form of the locale argument (which it expects to be a String value that is a well-formed BCP 47 language tag as verified by the IsWellFormedLanguageTag abstract operation). It takes the steps specified in RFC 5646 section 4.5, or successor, to bring the language tag into canonical form, and to regularize the case of the subtags. Implementations are allowed, but not required, to also canonicalize each extension subtag sequence within the tag according to the canonicalization specified by the standard registering the extension, such as RFC 6067 section 2.1.1.

NOTE RFC 5646 section 4.5 also provides steps to bring a language tag into “extlang form”, and allows the reordering of variant subtags. CanonicalizeLanguageTag does not take these steps.

6.3 Currency Codes

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

Implementations shall accept all well-formed 3-letter ISO 4217 currency codes. The set of combinations of currency code and language tag for which localized currency symbols are available is implementation dependent.

6.3.1 IsWellFormedCurrencyCode (currency)

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

1. Let *currencyCodeRE* be the result of creating a new object as if by the expression **`new RegExp()`**, where **`RegExp`** is the standard built-in constructor with that name, with the arguments **`"^[A-Z]{3}$"`** and **`"i"`**.
2. Return the result of calling the test method of *currencyCodeRE* with the argument *currency*.

6.4 Time Zone Names

This version of the ECMAScript Globalization API defines a single time zone name, "UTC", which identifies the UTC time zone.

Implementations shall support UTC and the host environment’s current time zone (if different from UTC).

7 The Globalization Object

The Globalization object is a single object that has some named properties, all of which are constructors.

The value of the [[Prototype]] internal property of the Globalization object is the built-in Object prototype object specified by the ECMAScript Language Specification. The value of the [[Extensible]] internal property is false.

NOTE The [[Extensible]] internal property is set to false for compatibility with the future module system in the ECMAScript Language Specification, 6 edition.

The Globalization object does not have a [[Construct]] internal property; it is not possible to use the Globalization object as a constructor with the new operator.

The Globalization object does not have a [[Call]] internal property; it is not possible to invoke the Globalization object as a function.

7.1 Constructor Properties of the Globalization Object

Each of the properties of the Globalization object is a constructor. The common behavior of these constructors is specified in this section; all remaining aspects are specified in the following clauses: LocaleList, Collator, NumberFormat, and DateTimeFormat.

7.1.1 Properties of the Constructors and Their Prototypes

Unless specified otherwise in this document, the constructor properties of the Globalization object and their prototypes shall have the properties and behavior specified for standard built-in ECMAScript objects in the ECMAScript Language Specification 5.1 edition, introduction of clause 15, or successor.

7.2 Internal Properties of the Globalization Object

The Globalization object has a [[currentHostLocale]] internal property, whose value is a String value representing the well-formed (6.2.1) and canonicalized (6.2.2) BCP 47 language tag for the host environment's current locale.

8 LocaleList Objects

LocaleList objects represent lists of language tags identifying locales. They can be used in two ways:

- To represent a language priority list, as described in RFC 4647, section 2.3, or successor. Algorithms interpreting a LocaleList object in this sense treat the list as ordered in descending order of priority.
- To represent a set of locales, such as those supported by an application or by the implementation of an object described in this specification. Algorithms interpreting a LocaleList object in this sense treat the list as unordered.

LocaleList objects have the properties of a generic array-like object: A length property and other properties whose names are array indices, as defined in the ECMAScript Language Specification, 5.1 edition, 15.4. The value of the length property is numerically greater than the name of every property inserted during construction whose name is an array index. As LocaleList objects are extensible, this invariant is not guaranteed to be maintained after construction.

The LocaleList constructor is a property of the Globalization object. Behavior common to all constructor properties of the Globalization object is specified in 7.1.

8.1 The LocaleList Constructor Called as a Function

When Globalization.LocaleList is called with a this value that is not an object whose constructor property is Globalization.LocaleList itself, it creates and initializes a new LocaleList object. Thus the function call Globalization.LocaleList(...) is equivalent to the object creation expression new Globalization.LocaleList(...) with the same arguments.

8.2 The LocaleList Constructor

When Globalization.LocaleList is called with a this value that is an object whose constructor property is Globalization.LocaleList itself, it acts as a constructor: it initializes the object.

8.2.1 new Globalization.LocaleList (locales)

When the LocaleList constructor is called with one argument, it interprets the locales argument as an array and copies its elements into the newly constructed object, validating the elements as well-formed language tags using the abstract operation IsWellFormedLanguageTag (6.2.1), and omitting duplicates.

1. Let *obj* be the **this** value.
2. Let *index* be 0.
3. Let *seen* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.
4. Let *cb* be a function that takes the argument *value* and performs the following steps:
    a. Let *tag* be ToString(*value*).
    b. If the result of calling the abstract operation IsWellFormedLanguageTag, passing *tag* as the argument, is **false**, then throw a **RangeError** exception.
    c. Replace *tag* with the result of calling the abstract operation CanonicalizeLanguageTag, passing *tag* as the argument.
    d. Let *duplicate* be the result of calling the [[HasProperty]] internal method of *seen* with argument *tag*.
    e. If *duplicate* is true, then return.
    f. Call the [[Put]] internal method of *seen* with arguments *tag*, **true**, and **true**.
    g. Let *desc* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.
    h. Call the [[Put]] internal method of *desc* with the arguments **`"value"`**, *tag*, and **true**.
    i. Call the [[Put]] internal method of *desc* with the arguments **`"enumerable"`**, **true**, and **true**.
    j. Call the Object.defineProperty function with arguments *obj*, ToString(*index*), and *desc*.
    k. Increase *index* by 1.
5. Apply Array.prototype.forEach to *locales* with argument *cb*.
6. Let *desc* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.
7. Call the [[Put]] internal method of *desc* with the arguments **`"value"`**, *index*, and **true**.
8. Call the Object.defineProperty function with arguments **this**, **`"length"`**, and *desc*.

The [[Prototype]] internal property of the newly constructed object is set to the original LocaleList prototype object, the one that is the initial value of LocaleList.prototype (8.3.1).

The [[Extensible]] internal property of the newly constructed object is set to true.

8.2.2 new Globalization.LocaleList ()

When the LocaleList constructor is called with no argument, it behaves as if it had received the array [locale] as the first argument, where locale is the value of the [[currentHostLocale]] internal property of the Globalization object.

8.3 Properties of the LocaleList Constructor

Besides the internal properties and the length property (whose value is 1), the LocaleList constructor has the following properties:

8.3.1 Globalization.LocaleList.prototype

The initial value of Globalization.LocaleList.prototype is the built-in LocaleList prototype object (8.4).

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

8.4 Properties of the LocaleList Prototype Object

The LocaleList prototype object is itself a LocaleList object, whose internal properties are set as if it had been constructed using new LocaleList([]).

In the following descriptions of functions that are properties of the LocaleList prototype object, the phrase “this LocaleList object” refers to the object that is the this value for the invocation of the function.

8.4.1 Globalization.LocaleList.prototype.constructor

The initial value of Globalization.LocaleList.prototype.constructor is the built-in LocaleList constructor.

8.4.2 [[IndexOfMatchFor]] (locale)

The [[IndexOfMatchFor]] internal method compares the provided argument *locale*, which it expects to be a String value with a well-formed and canonicalized BCP 47 language tag, against the locales in this locale list and returns the index of the best available match. It uses the fallback mechanism of RFC 4647, section 3.4. The following steps are taken:

1. Repeat while the value of the length property of *locale* is greater than 0:
    a. Let *index* be the result of applying the Array.prototype.indexOf method to this LocaleList object with the argument list `[`*locale*`]`.
    b. If *index* does not equal -1, then return *index*.
    c. Let *pos* be the result of calling the lastIndexOf method of *locale* with the argument `"-"`.
    d. If *pos* does not equal -1, then
        i. If *pos* is greater or equal to 2 and the character at index *pos*-2 of *locale* equals `"-"`, then decrease *pos* by 2.
        ii. Let *locale* be the result of calling the substring method of *locale* with arguments 0 and *pos*.
    e. Else let *locale* be `""`.
2. Return -1.

8.4.3 [[LookupMatch]] (requestedLocales)

The [[LookupMatch]] internal method compares *requestedLocales*, which it expects to be a LocaleList object representing a BCP 47 language priority list, against the set of locales in this LocaleList object, and determines the best available language to meet the request. The algorithm is based on the Lookup algorithm described in RFC 4647 section 3.4, but options specified through Unicode extension sequences are ignored in the lookup. Information about such subsequences is returned separately. The internal method returns an object with a locale property, whose value is the language tag of the selected locale, which must be an element of this LocaleList object. If the language tag of the request locale that led to the selected locale contained a Unicode extension subsequence, then the returned object also contains an extension property whose value is the Unicode extension subsequence (starting with "-u-"), and an extensionIndex property whose value is the index of the Unicode extension subsequence within the request locale language tag.

The following steps are taken:

1. Let *extensionMatch* be null.
2. Let *i* be 0.
3. Let *availableIndex* be -1.
4. Repeat while *i* is less than the value of the length property of *requestedLocales* and *availableIndex* is -1:
    a. Let *locale* be the element at index *i* of *requestedLocales*.
    b. Let *extensionMatch* be the result of calling the match method of *locale* with the argument `/-u(-([a-z0-9]{2,8}))+/`.
    c. If *extensionMatch* is not **null**, then:
        i. Let *extension* be the value of the element at index 0 of *extensionMatch*.
        ii. Let *extensionIndex* be the value of the index property of *extensionMatch*.
        iii. Let *locale* be the result of calling the replace method of *locale* with arguments *extension* and `""`.
    d. Let *availableIndex* be the result of calling the [[IndexOfMatchFor]] method of this LocaleList object, passing *locale* as the argument.
    e. Increase *i* by 1.
5. Let *result* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
6. If *availableIndex* does not equal -1, then:

a. Call the [[Put]] internal method of *result* with the arguments **"locale"**, the element at index *availableIndex* of this LocaleList object, and **true**.
b. If *extensionMatch* is not **null**, then:
   i. Call the [[Put]] internal method of result with the arguments **"extension"**, *extension*, and **true**.
   ii. Call the [[Put]] internal method of result with the arguments **"extensionIndex"**, *extensionIndex*, and **true**.

7. Else
   a. Call the [[Put]] internal method of result with the arguments **"locale"**, the value of the [[currentHostLocale]] internal property of the Globalization object, and **true**.
8. Return *result*.

8.4.4 [[BestFitMatch]] (requestedLocales)

The [[BestFitMatch]] internal method compares *requestedLocales*, which it expects to be a LocaleList object representing a BCP 47 language priority list, against the set of locales in this LocaleList object, 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 [[LookupMatch]] internal method. Options specified through Unicode extension sequences must be ignored by the algorithm. Information about such subsequences is returned separately. The internal method returns an object with a locale property, whose value is the language tag of the selected locale, which must be an element of this LocaleList object. If the language tag of the request locale that led to the selected locale contained a Unicode extension subsequence, then the returned object also contains an extension property whose value is the Unicode extension subsequence (starting with "-u-"), and an extensionIndex property whose value is the index of the Unicode extension subsequence within the request locale language tag.

8.4.5 [[LookupSupportedLocalesOf]] (requestedLocales)

The [[LookupSupportedLocalesOf]] internal method returns the subset of the provided BCP 47 language priority list for which this LocaleList object has a matching locale when using the BCP 47 Lookup algorithm. Locales appear in the same order in the returned list as in the input list. The following steps are taken:

1. If the constructor of *requestedLocales* is not Globalization.LocaleList, then replace *requestedLocales* with a new LocaleList object as if by the expression **`new Globalization.LocaleList(requestedLocales)`**, where **`Globalization.LocaleList`** is the standard built-in constructor with that name.
2. Let *callback* be a function that takes the argument *locale* and performs the following steps:
   a. Let *locale* be the result of calling the replace method of *locale* with the arguments **`/-u(-([a-z0-9]{2,8}))+/`** and **""**.
   b. Let *index* be the result of calling the [[IndexOfMatchFor]] internal method of this LocaleList object, passing *locale* as the argument.
   c. If *index* does not equal -1, then return **true**, otherwise return **false**.
3. Let *subset* be the result of applying the Array.prototype.filter method to *requestedLocales*, passing the argument list **`[`***callback*`, this]`**.
4. Return the result of creating a new object as if by the expression **`new Globalization.LocaleList(`***subset*`)`**, where **`Globalization.LocaleList`** is the standard built-in constructor with that name.

8.4.6 [[BestFitSupportedLocalesOf]] (requestedLocales)

The [[BestFitSupportedLocalesOf]] internal method returns the subset of the provided BCP 47 language priority list for which this LocaleList object has a matching locale when using the Best Fit Match algorithm. Locales appear in the same order in the returned list as in the input list. The steps taken are implementation dependent.

8.4.7 [[SupportedLocalesOf]] (requestedLocales, options)

The [[SupportedLocalesOf]] internal method returns the subset of the provided BCP 47 language priority list for which this LocaleList object has a matching. 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 the input list. The following steps are taken:

1. If *options* is provided and not **undefined**, then
    a. Let *matcher* be the result of calling the [[Get]] internal method of *options* with argument `"localeMatcher"`.
    b. If *matcher* is not **undefined**, then
        i. Let *matcher* be ToString(matcher).
        ii. If *matcher* is not equal to `"lookup"` or `"best fit"`, then throw a **RangeError** exception.
2. If *matcher* is **undefined** or equals `"best fit"` then
    a. Return the result of calling the [[BestFitSupportedLocalesOf]] internal method of this LocaleList object with argument *requestedLocales*.
3. Else
    a. Return the result of calling the [[LookupSupportedLocalesOf]] internal method of this LocaleList object with argument *requestedLocales*.

8.5 Properties of LocaleList Instances

LocaleList instances inherit properties from the LocaleList prototype object. LocaleList instances also have the following properties.

8.5.1 length

The length property of this LocaleList object is a data property whose value is a numeric value that is one greater than than the name of every property whose name is an array index.

The length property initially has the attributes { [[Writable]]: false, [[Enumerable]]: false, [[Configurable]]: false }.

8.5.2 Properties With Array Index Names

A LocaleList object has properties whose names are array indices from 0 to (length - 1). The value of each of these properties is a String value representing a well-formed language tag. The values are unique within a LocaleList object.

These properties initially have the attributes { [[Writable]]: false, [[Enumerable]]: true, [[Configurable]]: false }.

9 Locale and Parameter Negotiation

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

9.1 Internal Properties of Service Constructors

The constructors Collator, NumberFormat, and DateTimeFormat have the following internal properties:

- [[availableLocales]] is a LocaleList object with BCP 47 language tags identifying the locales for which the implementation provides the functionality of the constructed objects. The list must include the value of the [[currentHostLocale]] internal property of the Globalization object (7.2). Language tags on the list must not have a Unicode extension sequence.
- [[relevantExtensionKeys]] is an array 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 Collator) and [[localeData]] (for NumberFormat and DateTimeFormat) are objects that have properties for each locale contained in [[availableLocales]]. The value of each of these properties must be an object which has properties for each key contained in

[[relevantExtensionKeys]]. The value of each of these properties must be a non-empty array of those values defined in Unicode Technical Standard 35 for the given key that are supported by the implementation for the given locale, with the first element providing the default value.

EXAMPLE An implementation of DateTimeFormat might include the language tag "th" in its [[availableLocales]] internal property, and must (according to 12.3.3) include the key "ca" in its [[relevantExtensionKeys]] internal property. 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 property would therefore at least include {"th": {ca: ["buddhist", "gregory", "chinese", "islamicc"]}}.

NOTE Implementations should include in [[availableLocales]] locales that can serve as fallbacks in the algorithm used to resolve locales (see 9.2.1). For example, implementations shouldn't just provide a "de-DE" locale; they should include a "de" locale that can serve as a fallback for requests such as "de-AT" and "de-CH". For locales that in current usage would include a script subtag (such as Chinese locales), old-style language tags without script subtags should be included such that, for example, requests for "zh-TW" and "zh-HK" lead to output in traditional Chinese rather than the default simplified Chinese.

9.2 Abstract Operations

9.2.1 ResolveLocale (availableLocales, requestedLocales, options, relevantExtensionKeys, localeData)

The ResolveLocale abstract operation compares a BCP 47 language priority list against a set of available locales and determines the best available language to meet the request. 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 extension sequences are negotiated separately, taking the caller's relevant extension keys and locale data as well as client-provided options into consideration. The operation expects that the availableLocales argument is a LocaleList object. It returns an object with a locale property whose value is the language tag of the selected locale, and properties for each key in relevantExtensionKeys providing the selected value for that key.

The following steps are taken:

1. If the constructor of *requestedLocales* is not Globalization.LocaleList, then replace *requestedLocales* with a new LocaleList object as if by the expression **`new Globalization.LocaleList(requestedLocales)`**, where **`Globalization.LocaleList`** is the standard built-in constructor with that name.
2. Let *matcher* be the value of the localeMatcher property of *options*.
3. If *matcher* equals **`"lookup"`** then
    a. Let *r* be the result of calling the [[LookupMatch]] internal method of *availableLocales* with the argument *requestedLocales*.
4. Else
    a. Let *r* be the result of calling the [[BestFitMatch]] internal method of *availableLocales* with the argument *requestedLocales*.
5. Let *foundLocale* be the result of calling the [[Get]] internal method of *r* with the argument **`"locale"`**.
6. Let *extension* be the result of calling the [[Get]] internal method of *r* with the argument **`"extension"`**.
7. If *extension* is not **undefined**, then
    a. Let *extensionIndex* be the result of calling the [[Get]] internal method of *r* with the argument **`"extensionIndex"`**.
    b. Let *extensionSubtags* be the result of calling the split method of *extension* with the argument **`"-"`**.
8. Let *result* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.
9. Call the [[Put]] internal method of *result* with the arguments **`"dataLocale"`**, *foundLocale*, and **true**.
10. Let *supportedExtension* be **`"-u"`**.
11. Let *i* be 0.
12. Repeat while *i* is less then the value of the length property of *relevantExtensions*:
    a. Let *key* be the element at index *i* of *relevantExtensions*.
    b. Let *foundLocaleData* be the result of calling the [[Get]] internal method of *localeData* with the argument *foundLocale*.

c. Let *keyLocaleData* be the result of calling the [[Get]] internal method of *foundLocaleData* with the argument *key*.
d. Let *value* be the element at index 0 of *keyLocaleData*.
e. If *extensionSubtags* is not **undefined**, then
    i. Let *keyPos* be the result of calling the indexOf method of *extensionSubtags* with argument *key*.
    ii. If *keyPos* does not equal -1 and *keyPos* + 1 is less than or equal to the value of the length property of *extensionSubtags*, then
        1. Let *requestedValue* be the element at index *keyPos* + 1 of *extensionSubtags*.
        2. If the result of calling the indexOf method of *keyLocaleData* with the argument *requestedValue* is not -1, then
            a. Let *value* be *requestedValue*.
            b. Let *supportedExtension* be the concatenation of *supportedExtension*, `"-"`, *key*, `"-"`, and *value*.
f. If *options* has an own property with name *key*, and the result of calling the indexOf method of *keyLocaleData* with the value of the property named *key* of *options* is not -1, then let *value* be the value of the property named *key* of *options*.
g. Call the [[Put]] internal method of *result* with the arguments *key*, *value*, and **true**.
h. Increase *i* by 1.

13. If the value of the length property of *supportedExtension* is greater than 2, then
    a. Let *preExtension* be the result of calling the substring method of *foundLocale* with arguments 0 and *extensionIndex*.
    b. Let *postExtension* be the result of calling the substring method of *foundLocale* with argument *extensionIndex*.
    c. Let *foundLocale* be the concatenation of *preExtension*, *supportedExtension*, and *postExtension*.
14. Call the [[Put]] internal method of result with the arguments `"locale"`, *foundLocale*, and **true**.
15. Return *result*.

9.2.2 GetGetOption (options)

The GetGetOption abstract operation returns a function that extracts a property value 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.

When the GetGetOption abstract operation is called with argument *options*, the following steps are taken.

1. Let *getOption* be a function which, when called with arguments *property*, *type*, *values*, and *fallback*, takes the following steps:
    a. Let *value* be the result of calling the [[Get]] internal method of *options* with argument *property*.
    b. If *value* is neither **undefined** nor **null** then
        i. If *type* equals `"boolean"` then let *value* be ToBoolean(*value*).
        ii. If *type* equals `"string"` then let *value* be ToString(*value*).
        iii. If *type* equals `"number"` then let *value* be ToNumber(*value*).
        iv. If *values* is not **undefined**, then
            1. If the result of calling the indexOf method of *values* with argument *value* is -1, then throw a **RangeError** exception.
        v. Return *value*.
    c. Else return *fallback*.
2. Return *getOption*.

9.2.3 GetGetNumberOption (options)

The GetGetNumberOption abstract operation returns a function that extracts a property value 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.

When the GetGetNumberOption abstract operation is called with argument *options*, the following steps are taken.

1. Let *getNumberOption* be a function which, when called with arguments *property*, *minimum, maximum*, and *fallback*, takes the following steps:
    a. Let *value* be the result of calling the [[Get]] internal method of *options* with argument *property*.
    b. If *value* is neither **undefined** nor **null** then
        i. Let *value* be ToNumber(*value*).
        ii. If *value* is **NaN** or less than *minimum* or greater than *maximum*, throw a **RangeError** exception.
        iii. Return floor(*value*).
    c. Else return *fallback*.
2. Return *getNumberOption*.

10 Collator Objects

The Collator constructor is a property of the Globalization object. Behavior common to all service constructor properties of the Globalization object is specified in 7.1 and 9.1.

10.1 The Collator Constructor Called as a Function

When Globalization.Collator is called with a this value that is not an object whose constructor property is Globalization.Collator itself, it creates and initializes a new Collator object. Thus the function call Globalization.Collator(...) is equivalent to the object creation expression new Globalization.Collator(...) with the same arguments.

10.2 The Collator Constructor

When Globalization.Collator is called with a this value that is an object whose constructor property is Globalization.Collator itself, it acts as a constructor: it initializes the object.

10.2.1 new Globalization.Collator ([localeList [, options]])

When the Collator constructor is called with two arguments, it computes its effective locale and its collation options from these arguments. The computation uses the abstract operations ResolveLocale (9.2.1) and GetGetOption (9.2.2).

Several steps in the algorithm use values from the following table, which associates Unicode extension keys, property names, types, and allowable values:

Table 1 – Collator options settable through both extension keys and options properties

Key	Property	Type	Values
kb	backwards	"boolean"	
kc	caseLevel	"boolean"	
kn	numeric	"boolean"	
kh	hiraganaQuaternary	"boolean"	
kk	normalization	"boolean"	
kf	caseFirst	"string"	"upper", "lower", "false"

The following steps are taken:

1. If *localeList* is not provided or is **undefined**, then let *localeList* be the result of creating a new LocaleList object as if by the expression `new Globalization.LocaleList()` where `Globalization.LocaleList` is the standard built-in constructor with that name.

2. If *options* is not provided or is **undefined**, then let *options* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
3. Let *getOption* be the result of calling the GetGetOption abstract operation with argument *options*.
4. Let *u* be the result of calling *getOption* with arguments `"usage"`, `"string"`, `["sort", "search"]`, and `"sort"`.
5. Set the [[usage]] internal property of the newly constructed object to *u*.
6. If *u* is equal to `"sort"`, then let *localeData* be the value of the [[sortLocaleData]] internal property of Collator; else let *localeData* be the value of the [[searchLocaleData]] internal property of Collator.
7. Let *opt* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
8. Let *matcher* be the result of calling *getOption* with the arguments `"localeMatcher"`, `"string"`, `["lookup", "best fit"]`, and `"best fit"`.
9. Call the [[Put]] internal method of *opt* with arguments `"localeMatcher"`, *matcher*, and **true**.
10. For each row in Table 1, except the header row, do:
    a. Let *value* be the result of calling *getOption*, passing as arguments the name given in the Property column of the row, the string given in the Type column of the row, and, if the Values column of the row contains strings, an array containing those strings.
    b. Call the [[Put]] internal method of *opt*, passing as arguments the name given in the Key column of the row, *value*, and **true**.
11. Let *relevantExtensions* be the value of the [[relevantExtensionKeys]] internal property of Collator.
12. Let *r* be the result of calling the ResolveLocale abstract operation with the [[availableLocales]] internal property of Collator, the *localeList* argument, *opt*, *relevantExtensions*, and *localeData*.
13. Set the [[locale]] internal property of the newly constructed object to the value of the locale property of *r*.
14. Let *i* be 0.
15. Let *len* be the result of calling the [[Get]] internal method of *relevantExtensions* with argument `"length"`.
16. Repeat while *i* is less than *len*:
    a. Let *key* be the element at index *i* of *relevantExtensions*.
    b. If *key* is equal to `"co"`, then
        i. Let *property* be `"collation"`.
        ii. Let *value* be the value of the property co of *r*.
        iii. If *value* is **null**, then let *value* be `"default"`.
    c. Else use the row of Table 1 that contains the value of *key* in the Key column:
        i. Let *property* be the name given in the Property column of the row.
        ii. Let *value* be the value of the property named *key* of *r*.
        iii. If the name given in the Type column of the row is `"boolean"`, then let *value* be the result of comparing *value* with `"true"`.
    d. Set the internal property named *property* of the newly constructed object to *value*.
    e. Increase *i* by 1.
17. Let *s* be the result of calling *getOption* with arguments `"sensitivity"`, `"string"`, and `["base", "accent", "case", "variant"]`.
18. If *s* is **undefined**, then
    a. If *u* is equal to `"sort"`, then let s be `"variant"`.
    b. Else
        i. Let *dataLocale* be the value of the dataLocale property of *r*.
        ii. Let *s* be the value of the sensitivity property of the value of the property named *dataLocale* of *localeData*.
19. Set the [[sensitivity]] internal property of the newly constructed object to *s*.
20. Let *ip* be the result of calling *getOption* with arguments `"ignorePunctuation"`, `"boolean"`, **undefined**, and **false**.
21. Set the [[ignorePunctuation]] internal property of the newly constructed object to *ip*.

The [[Prototype]] internal property of the newly constructed object is set to the original Collator prototype object, the one that is the initial value of Collator.prototype (10.3.1).

The [[Extensible]] internal property of the newly constructed object is set to true.

10.3 Properties of the Collator Constructor

Besides the internal properties and the length property (whose value is 2), the Collator constructor has the following properties:

10.3.1 Globalization.Collator.prototype

The initial value of Globalization.Collator.prototype is the built-in Collator prototype object (10.4).

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

10.3.2 Globalization.Collator.supportedLocalesOf (requestedLocales [, options])

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

1. Let *availableLocales* be the value of the [[availableLocales]] internal property of Collator.
2. Return the result of calling the [[SupportedLocalesOf]] internal method of *availableLocales* with the arguments *requestedLocales* and *options*.

10.3.3 Internal Properties

The initial value of the [[availableLocales]] internal property is implementation dependent within the constraints described in 9.1.

The initial value of the [[relevantExtensionKeys]] internal property is an array that must include the element "co", may include any or all of the elements "kb", "kc", "kn", "kh", "kk", "kf", and must not include any other elements.

NOTE Unicode Technical Standard 35 describes ten locale extension keys that are relevant to collation: "co" for collator usage and specializations, "ka" for alternate handling, "kb" for backward second level weight, "kc" for case level, "kn" for numeric, "kh" for hiragana quaternary, "kk" for normalization, "kf" for case first, "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 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 key "vt" is not supported in this version of the Globalization API. Support for the remaining keys is implementation dependent.

The initial values of the [[sortLocaleData]] and [[searchLocaleData]] internal properties are implementation dependent within the constraints described in 9.1 and the following additional constraints:

- The first element of [[sortLocaleData]][locale].co and [[searchLocaleData]][locale].co must be null for all locale values.
- The values "standard" and "search" must not be used as elements in any [[sortLocaleData]][locale].co and [[searchLocaleData]][locale].co array.
- [[searchLocaleData]][locale] must have a sensitivity property with a String value equal to "base", "accent", "case", or "variant" for all locale values.

10.4 Properties of the Collator Prototype Object

The Collator prototype object is itself a Collator object, whose internal properties are set as if it had been constructed using new Collator().

In the following descriptions of functions that are properties of the Collator prototype object, the phrase “this Collator object” refers to the object that is the this value for the invocation of the function.

10.4.1 Globalization.Collator.prototype.constructor

The initial value of Globalization.Collator.prototype.constructor is the built-in Collator constructor.

10.4.2 Globalization.Collator.prototype.compare (x, y)

When the compare method is called with two arguments x and y, it returns a Number other than NaN that represents the result of a locale-sensitive String comparison of x (converted to a String) with y (converted to a String). The two Strings are X and 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 this Collator object, and will be negative, zero, or positive, depending on whether X comes before Y in the sort order, the Strings are equal under the sort order, or X comes after Y in the sort order, respectively.

The sensitivity of the collator is interpreted as follows:

- base: Only strings that differ in base letters compare as unequal. Examples: a ≠ b, a = á, a = A, ぁ = あ.
- accent: Only strings that differ in base letters or accents compare as unequal. Examples: a ≠ b, a ≠ á, a = A, ぁ = あ.
- case: Only strings that differ in base letters or case compare as unequal. Examples: a ≠ b, a = á, a ≠ A, ぁ = あ.
- variant: Strings that differ in base letters, accents, case, or width compare as unequal. Examples: a ≠ b, a ≠ á, a ≠ A, ぁ ≠ あ.

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.

Before performing the comparison, the following steps are performed to prepare the Strings:

1. Let X be ToString(x).
2. Let Y be ToString(y).

The compare method of any given Collator object, if considered as a function of two arguments x and y, is a consistent comparison function (as defined in the ECMAScript Language Specification, 5.1 edition, 15.4.4.11, or successor) on the set of all Strings.

The actual return values are implementation-defined to permit implementers to encode additional information in the value, but the method is required to define a total ordering on all Strings and to return 0 when comparing Strings that are considered canonically equivalent by the Unicode standard.

NOTE 1 The compare method itself is not directly suitable as an argument to Array.prototype.sort because it must be invoked as the method of a Collator object.

NOTE 2 It is recommended that the compare method be implemented following Unicode Technical Standard #10, Unicode Collation Algorithm, using tailorings for the effective locale and collation options of this Collator object.

NOTE 3 Applications should not assume that the behavior of the compare methods of Collator instances with the same resolved options will remain the same for different versions of the same implementation.

10.4.3 Globalization.Collator.prototype.resolvedOptions

This named accessor property provides access to the locale and collation options computed during initialization of the object. The value of the [[Get]] attribute is a function that returns a new object with properties locale, usage, sensitivity, ignorePunctuation, collation, as well as those properties shown in Table 1 whose keys are included in the [[relevantExtensionKeys]] internal property of Collator. Each property has the value of the corresponding internal property of this Collator object (see 10.5). The [[Set]] attribute is undefined.

10.5 Properties of Collator Instances

Collator instances inherit properties from the Collator prototype object. Collator instances also have several internal properties 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.

Collator instances also have the following internal properties if the key corresponding to the name of the internal property in Table 1 is included in the [[relevantExtensionKeys]] internal property of Collator:

- [[backwards]] is a Boolean value, specifying whether backward second level weight is used.
- [[caseLevel]] is a Boolean value, specifying whether case level adjustment is used.
- [[numeric]] is a Boolean value, specifying whether numeric sorting is used.
- [[hiraganaQuaternary]] is a Boolean value, specifying whether hiragana characters receive special treatment at quaternary level.
- [[normalization]] is a Boolean value, specifying whether strings should be normalized before comparison.
- [[caseFirst]] is a String value; allowed values are specified in Table 1.

11 NumberFormat Objects

The NumberFormat constructor is a property of the Globalization object. Behavior common to all service constructor properties of the Globalization object is specified in 7.1 and 9.1.

11.1 The NumberFormat Constructor Called as a Function

When Globalization.NumberFormat is called with a this value that is not an object whose constructor property is Globalization.NumberFormat itself, it creates and initializes a new NumberFormat object. Thus the function call Globalization.NumberFormat(...) is equivalent to the object creation expression new Globalization.NumberFormat(...) with the same arguments.

11.2 The NumberFormat Constructor

When Globalization.NumberFormat is called with a this value that is an object whose constructor property is Globalization.NumberFormat itself, it acts as a constructor: it initializes the object.

11.2.1 new Globalization.NumberFormat ([localeList [, options]])

When the NumberFormat constructor is called with arguments localeList and options, it computes its effective locale and its formatting options from these arguments. The computation uses the abstract operations ResolveLocale (9.2.1), GetGetOption (9.2.2), GetGetNumberOption (9.2.3), and CurrencyDigits (defined below).

The following steps are taken:

1. If *localeList* is not provided or is **undefined**, then let *localeList* be the result of creating a new LocaleList object as if by the expression **`new Globalization.LocaleList()`** where **`Globalization.LocaleList`** is the standard built-in constructor with that name.
2. If *options* is not provided or is **undefined**, then let *options* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.
3. Let *getOption* be the result of calling the GetGetOption abstract operation with argument *options*.
4. Let *getNumberOption* be the result of calling the GetGetNumberOption abstract operation with argument *options*.
5. Let *opt* be the result of creating a new object as if by the expression **`new Object()`** where **`Object`** is the standard built-in constructor with that name.

6. Let *matcher* be the result of calling *getOption* with the arguments **"localeMatcher"**, **"string"**, **["lookup", "best fit"]**, and **"best fit"**.
7. Call the [[Put]] internal method of *opt* with arguments **"localeMatcher"**, *matcher*, and **true**.
8. Let *localeData* be the value of the [[localeData]] internal property of DateTimeFormat.
9. Let *r* be the result of calling the ResolveLocale abstract operation with the [[availableLocales]] internal property of NumberFormat, the *localeList* argument, *opt*, the [[relevantExtensionKeys]] internal property of NumberFormat, and *localeData*.
10. Set the [[locale]] internal property of the newly constructed object to the value of the locale property of *r*.
11. Set the [[numberingSystem]] internal property of the newly constructed object to the value of the nu property of *r*.
12. Let *dataLocale* be the value of the dataLocale property of *r*.
13. Let *s* be the result of calling *getOption* with the arguments **"style"**, **"string"**, **["decimal", "percent", "currency"]**, and **"decimal"**.
14. Set the [[style]] internal property of the newly constructed object to *s*.
15. Let *c* be the result of calling *getOption* with the arguments **"currency"** and **"string"**.
16. If *c* is not undefined and the result of calling the IsWellFormedCurrencyCode abstract operation with argument *c* is false, then throw a **RangeError** exception.
17. If *s* is equal to **"currency"** and *c* is **undefined**, throw a **TypeError** exception.
18. If *s* is equal to **"currency"** then set the [[currency]] internal property of the newly constructed object to *c*, converted to upper case as specified in 6.1.
19. Let *cd* be the result of calling *getOption* with the arguments **"currencyDisplay"**, **"string"**, **["code", "symbol", "name"]**, and **"symbol"**.
20. If *s* is equal to **"currency"** then set the [[currencyDisplay]] internal property of the newly constructed object to *cd*.
21. Let *mnid* be the result of calling *getNumberOption* with the arguments **"minimumIntegerDigits"**, 1, 21, and 1.
22. Set the [[minimumIntegerDigits]] internal property of the newly constructed object to *mnid*.
23. If *s* is equal to **"currency"** then let *mnfdDefault* be CurrencyDigits(*c*); else let *mnfdDefault* be 0.
24. Let *mnfd* be the result of calling *getNumberOption* with the arguments **"minimumFractionDigits"**, 0, 20, and *mnfdDefault*.
25. Set the [[minimumFractionDigits]] internal property of the newly constructed object to *mnfd*.
26. If *s* is equal to **"currency"** then let *mxfdDefault* be max(*mnfd*, CurrencyDigits(*c*)); else if *s* is equal to **"percent"** then let *mxfdDefault* be max(*mnfd*, 0); else let *mxfdDefault* be max(*mnfd*, 3).
27. Let *mxfd* be the result of calling *getNumberOption* with the arguments **"maximumFractionDigits"**, *mnfd*, 20, and *mxfdDefault*.
28. Set the [[maximumFractionDigits]] internal property of the newly constructed object to *mxfd*.
29. Delete the [[minimumSignificantDigits]] and [[maximumSignificantDigits]] internal properties of the newly constructed object.
30. If *options* has at least one of the properties minimumSignificantDigits and maximumSignificantDigits, then:
    a. Let *mnsd* be the result of calling *getNumberOption* with the arguments **"minimumSignificantDigits"**, 1, 21, and 1.
    b. Let *mxsd* be the result of calling *getNumberOption* with the arguments **"maximumSignificantDigits"**, *mnsd*, 21, and 21.
    c. Set the [[minimumSignificantDigits]] internal property of the newly constructed object to *mnsd*, and the [[maximumSignificantDigits]] internal property of the newly constructed object to *mxsd*.
31. Let *g* be the result of calling *getOption* with the arguments **"useGrouping"**, **"boolean"**, **undefined**, and **true**.
32. Set the [[useGrouping]] internal property of the newly constructed object to *g*.
33. Let *patterns* be the value of the patterns property of the property named *dataLocale* of *localeData*.
34. Let *patterns* be the value of the property named *s* of *patterns*.
35. Set the [[positivePattern]] internal property of the newly constructed object to the value of the positivePattern property of *patterns*.
36. Set the [[negativePattern]] internal property of the newly constructed object to the value of the negativePattern property of *patterns*.

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

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

The [[Prototype]] internal property of the newly constructed object is set to the original NumberFormat prototype object, the one that is the initial value of NumberFormat.prototype (11.3.1).

The [[Extensible]] internal property of the newly constructed object is set to true.

11.3 Properties of the NumberFormat Constructor

Besides the internal properties and the length property (whose value is 2), the NumberFormat constructor has the following properties:

11.3.1 Globalization.NumberFormat.prototype

The initial value of Globalization.NumberFormat.prototype is the built-in NumberFormat prototype object (11.4).

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

11.3.2 Globalization.NumberFormat.supportedLocalesOf (requestedLocales [, options])

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

1. Let *availableLocales* be the value of the [[availableLocales]] internal property of NumberFormat.
2. Return the result of calling the [[SupportedLocalesOf]] internal method of *availableLocales* with the arguments *requestedLocales* and *options*.

11.3.3 Internal Properties

The initial value of the [[availableLocales]] internal property is implementation dependent within the constraints described in 9.1.

The initial value of the [[relevantExtensionKeys]] internal property is ["nu"].

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

The initial value of the [[ocaleData]] internal property is implementation dependent within the constraints described in 9.1 and the following additional constraints:

- [[localeData]][locale] must have a patterns property for all locale values. The value of this property must be an object, which must properties with the names of the three number format styles: `"decimal"`, `"percent"`, and `"currency"`. Each of these properties in turn must be objects with the properties positivePattern and negativePattern. The value of these properties must be string values which contain a substring `"{number}"`; the values within the currency property must also contain a substring `"{currency}"`.

11.4 Properties of the NumberFormat Prototype Object

The NumberFormat prototype object is itself a NumberFormat object, whose internal properties are set as if it had been constructed using new NumberFormat().

In the following descriptions of functions that are properties of the NumberFormat prototype object, the phrase “this NumberFormat object” refers to the object that is the this value for the invocation of the function.

11.4.1 Globalization.NumberFormat.prototype.constructor

The initial value of Globalization.NumberFormat.prototype.constructor is the built-in NumberFormat constructor.

11.4.2 Globalization.NumberFormat.prototype.format (value)

Returns a String value representing the result of calling ToNumber(value) according to the effective locale and the formatting options of this NumberFormat object.

The computations rely on String values and locations within numeric strings that are implementation and locale dependent ("ILD") or implementation, locale, and numbering system dependent ("ILND"). The ILD and ILND Strings mentioned must not contain digits as specified by the Unicode Standard.

The following steps are taken:

1. Let *x* be ToNumber(*value*).
2. Let *negative* be **false**.
3. If the result of isFinite(*x*) is **false**, then
    a. If *x* is **NaN**, then let *n* be an ILD String value indicating the NaN value.
    b. Else
        i. Let *n* be an ILD String value indicating infinity.
        ii. If *x* < 0, then let *negative* be **true**.
4. Else
    a. If *x* < 0, then
        i. Let *negative* be **true**.
        ii. Let *x* be -*x*.
    b. If the value of the [[style]] internal property of this NumberFormat object is **"percent"**, let *x* be 100 × *x*.
    c. If the [[minimumSignificantDigits]] and [[maximumSignificantDigits]] internal properties of this NumberFormat object are present, then
        i. Let *n* be the result of calling ToRawPrecision, passing as arguments *x* and the values of the [[minimumSignificantDigits]] and [[maximumSignificantDigits]] internal properties of this NumberFormat object.
    d. Else
        i. Let *n* be the result of calling ToRawFixed, passing as arguments *x* and the values of the [[minimumIntegerDigits]], [[minimumFractionDigits]], and [[maximumFractionDigits]] internal properties of this NumberFormat object.
    e. If the value of the [[numberingSystem]] internal property of this NumberFormat object matches one of the values in the "Numbering System" column of Table 2 below, then
        i. Let *digits* be a String value containing exactly the 10 digits specified in the "Digits" column of Table 2 in the row containing the value of the [[numberingSystem]] internal property.
        ii. Replace each *digit* in *n* with the value of *digits*[*digit*].
    f. Else use an implementation dependent algorithm to map *n* to the appropriate representation of *n* in the given numbering system.
    g. If *n* contains the character **"."**, then replace it with an ILND String representing the decimal separator.
    h. If the value of the [[useGrouping]] internal property of this NumberFormat object is **true**, then insert an ILND String representing a grouping separator into an ILND set of locations within the integer part of *n*.
5. If *negative* is **true**, then let *result* be the value of the [[negativePattern]] internal property of this NumberFormat object; else let *result* be the value of the [[positivePattern]] internal property of this NumberFormat object.
6. Replace the substring **"{number}"** within *result* with *n*.
7. If the value of the [[style]] internal property of this NumberFormat object is **"currency"**, then:
    a. Let *currency* be the value of the [[currency]] internal property of this NumberFormat object.
    b. If the value of the [[currencyDisplay]] internal property of this NumberFormat object is **"code"**, then let *cd* be *currency*.
    c. Else if the value of the [[currencyDisplay]] internal property of this NumberFormat object is **"symbol"**, then let *cd* be an ILD string representing *currency* in short form. If the implementation does not have such a representation of *currency*, then use *currency* itself.
    d. Else if the value of the [[currencyDisplay]] internal property of this NumberFormat object is **"name"**, then let *cd* be an ILD string representing *currency* in long form. If the implementation does not have such a representation of *currency*, then use *currency* itself.
    e. Replace the substring **"{currency}"** within *result* with *cd*.
8. Return *result*.

When the ToRawPrecision abstract operation is called with arguments *x* (expected to be a finite positive number), *minPrecision*, and *maxPrecision* (both expected to be integers between 1 and 21) the following steps are taken:

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

When the ToRawFixed abstract operation is called with arguments *x* (expected to be a finite positive number), *minInteger* (expected to be an integer between 1 and 21), *minFraction*, and *maxFraction* (both expected to 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. If *n* = 0, let *m* be the String `"0"`. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If *f* ≠ 0, then
    a. Let *k* be the number of characters in *m*.
    b. If *k* ≤ *f*, then
        i. Let *z* be the String consisting of *f*+1–*k* occurrences of the character `"0"`.
        ii. Let *m* be the concatenation of Strings *z* and *m*.
        iii. Let *k*=*f*+1.
    c. Let *a* be the first *k*–*f* characters of m, and let *b* be the remaining *f* characters of *m*.
    d. Let *m* be the concatenation of the three Strings *a*, ".", and *b*.
    e. Let *int* be the number of characters in *a*.
5. Else let *int* be the number of characters in *m*.
6. Let *cut* be *maxFraction* – *minFraction*.
7. Repeat while *cut* > 0 and the last character of *m* is `"0"`:
    a. Remove the last character from *m*.
    b. Decrease *cut* by 1.
8. If the last character of *m* is `"."`, then
    a. Remove the last character from *m*.
9. If int < *minInteger*, then
    a. Let *z* be the String consisting of *minInteger*–*int* occurrences of the character `"0"`.

b. Let *m* be the concatenation of Strings *z* and *m*.

10. Return *m*.

Table 2 – Numbering systems with simple digit mappings

Numbering System	Digits
arab	U+0660 to U+0669
arabext	U+06F0 to U+06F9
beng	U+09E6 to U+09EF
deva	U+0966 to U+096F
fullwide	U+FF10 to U+FF19
gujr	U+0AE6 to U+0AEF
guru	U+0A66 to U+0A6F
hanidec	U+3007, U+4E00, U+4E8C, U+4E09, U+56DB, U+4E94, U+516D, U+4E03, U+516B, U+4E5D
khmr	U+17E0 to U+17E9
knda	U+0CE6 to U+0CEF
laoo	U+0ED0 to U+0ED9
latn	U+0030 to U+0039
mlym	U+0D66 to U+0D6F
mong	U+1810 to U+1819
mymr	U+1040 to U+1049
orya	U+0B66 to U+0B6F
tamldec	U+0BE6 to U+0BEF
telu	U+0C66 to U+0C6F
thai	U+0E50 to U+0E59
tibt	U+0F20 to U+0F29

11.4.3 Globalization.NumberFormat.prototype.resolvedOptions

This named accessor property provides access to the locale and formatting options computed during initialization of the object. The value of the [[Get]] attribute is a function that returns a new object with properties locale, numberingSystem, style, currency, currencyDisplay, minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, maximumSignificantDigits, and useGrouping, each with the value of the corresponding internal property of this NumberFormat object (see 11.5). The [[Set]] attribute is undefined.

11.5 Properties of NumberFormat Instances

NumberFormat instances inherit properties from the NumberFormat prototype object. NumberFormat instances also have several internal properties that are computed by the constructor:

- [[locale]] is a String value with the language tag of the locale whose localization is used for formatting.
- [[numberingSystem]] is a String value with the "type" given in Unicode Technical Standard 35 for the numbering system used for formatting.
- [[style]] is one of the String values "decimal", "currency", or "percent", identifying the number format style used.
- [[currency]] is a String value with the currency code identifying the currency to be used if formatting with the "currency" style.

- [[currencyDisplay]] is one of the String values "code", "symbol", or "name", specifying whether to display the currency as an ISO 4217 alphabetic currency code, a localized currency symbol, or a localized currency name if formatting with the "currency" style.
- [[minimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used. Numbers will be padded with leading zeroes if necessary.
- [[minimumFractionDigits]] and [[maximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.
- [[minimumSignificantDigits]] and [[maximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be shown. Either none or both of these properties are defined; if they are, they override minimum and maximum integer and fraction digits – the formatter uses however many integer and fraction digits are required to display the specified number of significant digits.
- [[useGrouping]] is a Boolean value indicating whether a grouping separator should be used.
- [[positivePattern]] and [[negativePattern]] are String values as described in 11.3.3.

12 DateTimeFormat Objects

The DateTimeFormat constructor is a property of the Globalization object. Behavior common to all service constructor properties of the Globalization object is specified in 7.1 and 9.1.

12.1 The DateTimeFormat Constructor Called as a Function

When Globalization.DateTimeFormat is called with a this value that is not an object whose constructor property is Globalization.DateTimeFormat itself, it creates and initializes a new DateTimeFormat object. Thus the function call Globalization.DateTimeFormat(...) is equivalent to the object creation expression new Globalization.DateTimeFormat(...) with the same arguments.

12.2 The DateTimeFormat Constructor

When Globalization.DateTimeFormat is called with a this value that is an object whose constructor property is Globalization.DateTimeFormat itself, it acts as a constructor: it initializes the object.

12.2.1 new Globalization.DateTimeFormat ([localeList [, options]])

When the DateTimeFormat constructor is called with arguments localeList and options, it computes its effective locale and its formatting options from these arguments. The computation uses the abstract operations ResolveLocale (9.2.1), GetGetOption (9.2.2), ToDateTimeOptions, BasicFormatMatch, and BestFitFormatMatch (defined below). Several algorithms use values from the following table, which provides property names and allowable values for the components of date and time formats:

Table 3 – Components of date and time formats

Property	Values
weekday	"narrow", "short", "long"
era	"narrow", "short", "long"
year	"2-digit", "numeric"
month	"2-digit", "numeric", "narrow", "short", "long"
day	"2-digit", "numeric"
hour	"2-digit", "numeric"
minute	"2-digit", "numeric"
second	"2-digit", "numeric"
timeZoneName	"short", "long"

The following steps are taken:

1. If *localeList* is not provided or is **undefined**, then let *localeList* be the result of creating a new LocaleList object as if by the expression `new Globalization.LocaleList()` where `Globalization.LocaleList` is the standard built-in constructor with that name.
2. Let *options* be the result of calling the ToDateTimeOptions abstract operation with arguments *options*, **true**, and **false**.
3. Let *getOption* be the result of calling the GetGetOption abstract operation with argument *options*.
4. Let *opt* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
5. Let *matcher* be the result of calling *getOption* with the arguments `"localeMatcher"`, `"string"`, `["lookup", "best fit"]`, and `"best fit"`.
6. Call the [[Put]] internal method of *opt* with arguments `"localeMatcher"`, *matcher*, and **true**.
7. Let *localeData* be the value of the [[localeData]] internal property of DateTimeFormat.
8. Let *r* be the result of calling the ResolveLocale abstract operation with the [[availableLocales]] internal property of DateTimeFormat, the *localeList* argument, *opt*, the [[relevantExtensionKeys]] internal property of DateTimeFormat, and *localeData*.
9. Set the [[locale]] internal property of the newly constructed object to the value of the locale property of *r*.
10. Set the [[calendar]] internal property of the newly constructed object to the value of the ca property of *r*.
11. Set the [[numberingSystem]] internal property of the newly constructed object to the value of the nu property of *r*.
12. Let *dataLocale* be the value of the dataLocale property of *r*.
13. Let *tz* be the result of calling the [[Get]] internal method of *options* with argument `"timeZone"`.
14. If *tz* is not **undefined**, then
    a. Let *tz* be ToString(*tz*).
    b. Convert *tz* to upper case as described in 6.1.
    c. If *tz* is not equal to `"UTC"`, then throw a **RangeError** exception.
15. Set the [[timeZone]] internal property of the newly constructed object to *tz*.
16. Let *hr12* be the result of calling *getOption* with the arguments `"hour12"` and `"boolean"`.
17. If *hr12* is undefined, then let *hr12* be the value of the hour12 property of the property named *dataLocale* of *localeData*.
18. Set the [[hour12]] internal property of the newly constructed object to *hr12*.
19. Let *opt* be the result of creating a new object as if by the expression `new Object()` where `Object` is the standard built-in constructor with that name.
20. For each row of Table 3, except the header row, do:
    d. Let *value* be the result of calling *getOption*, passing as arguments the name given in the Property column of the row, `"string"`, and an array with the strings given in the Values column of the row.
    e. Call the [[Put]] internal method of *opt*, passing as arguments the name given in the Property column of the row, *value*, and **true**.
21. Let *formats* be the value of the formats property of the property named *dataLocale* of *localeData*.
22. Let *match* be the result of calling *getOption* with the arguments `"formatMatch"`, `"string"`, `["basic", "best fit"]`, and `"best fit"`.
23. If *match* equals `"basic"` then

f. Let *bestFormat* be the result of calling BasicFormatMatch with *opt* and *formats*.

24. Else
    g. Let *bestFormat* be the result of calling BestFitFormatMatch with *opt* and *formats*.
25. For each row in Table 3, except the header row, do
    h. Let *p* be the value of the property of *bestFormat* with the name given in the Property column of the row.
    i. If *p* is not undefined, then set the internal property of the newly constructed object with the name given in the Property column of the row to *p*.
26. Set the [[pattern]] internal property of the newly constructed object to the value of the pattern property of *bestFormat*.

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

1. If *options* is not supplied or is **undefined**, let options be **null**.
2. Let *options* be the result of calling the built-in function Object.create with argument *options*.
3. Let *needDefault* be **true**.
4. If *date* equals **true**, then
    a. For each of the property names **"weekday"**, **"year"**, **"month"**, **"day"**:
        i. If the result of calling the [[Get]] internal method of *options* with the property name is not **undefined**, then let *needDefault* be **false**.
5. If *time* equals **true**, then
    a. For each of the property names **"hour"**, **"minute"**, **"second"**:
        i. If the result of calling the [[Get]] internal method of *options* with the property name is not **undefined**, then let *needDefault* be **false**.
6. If *needDefault* and *date* both equal **true**, then
    a. For each of the property names **"year"**, **"month"**, **"day"**:
        i. Call the [[Put]] internal method of *options* with the property name, **"numeric"**, and **true**.
7. If *needDefault* and *time* both equal **true**, then
    a. For each of the property names **"hour"**, **"minute"**, **"second"**:
        i. Call the [[Put]] internal method of *options* with the property name, **"numeric"**, and **true**.
8. Return *options*.

When the BasicFormatMatch 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 *shortMorePenalty* be 3.
4. Let *shortLessPenalty* be 6.
5. Let *longMorePenalty* be 6.
6. Let *longLessPenalty* be 8.
7. Let *bestScore* be **-Infinity**.
8. Let *bestFormat* be **undefined**.
9. Let *i* be 0.
10. Let *len* be the value of the length property of formats:
11. Repeat while *i* is less than *len*:
    a. Let *format* be the element at index *i* of *formats*.
    b. Let *score* be 0.
    c. For each *property* shown in Table 3:
        i. Let *optionsProp* be the value of the property named *property* of *options*.
        ii. Let *formatProp* be the value of the property named *property* of *format*.
        iii. If *optionsProp* is **undefined** and *formatProp* is not **undefined**, then decrease *score* by *additionPenalty*.
        iv. Else if *optionsProp* is not **undefined** and *formatProp* is **undefined**, then decrease *score* by *removalPenalty*.
        v. Else
            1. Let *values* be the array **["2-digit", "numeric", "narrow", "short", "long"]**.
            2. Let *optionsPropIndex* be the index of *optionsProp* within *values*.

3. Let *formatPropIndex* be the index of *formatProp* within *values*.
4. Let *delta* be max(min(*formatPropIndex* - *optionsPropIndex*, 2), -2).
5. If *delta* equals 2, decrease *score* by *longMorePenalty*.
6. Else if *delta* equals 1, decrease *score* by *shortMorePenalty*.
7. Else if *delta* equals -1, decrease *score* by *shortLessPenalty*.
8. Else if *delta* equals -2, decrease *score* by *longLessPenalty*.

d. If *score* is greater than *bestScore*, then
   i. Let *bestScore* be *score*.
   ii. Let *bestFormat* be *format*.

e. Increase *i* by 1.

12. Return *bestFormat*.

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

The [[Prototype]] internal property of the newly constructed object is set to the original DateTimeFormat prototype object, the one that is the initial value of DateTimeFormat.prototype (12.3.1).

The [[Extensible]] internal property of the newly constructed object is set to true.

12.3 Properties of the DateTimeFormat Constructor

Besides the internal properties and the length property (whose value is 2), the DateTimeFormat constructor has the following properties:

12.3.1 Globalization.DateTimeFormat.prototype

The initial value of Globalization.DateTimeFormat.prototype is the built-in DateTimeFormat prototype object (12.4).

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

12.3.2 Globalization.DateTimeFormat.supportedLocalesOf (requestedLocales [, options])

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

1. Let *availableLocales* be the value of the [[availableLocales]] internal property of DateTimeFormat.
2. Return the result of calling the [[SupportedLocalesOf]] internal method of *availableLocales* with the arguments *requestedLocales* and *options*.

12.3.3 Internal Properties

The initial value of the [[availableLocales]] internal property is implementation dependent within the constraints described in 9.1.

The initial value of the [[relevantExtensionKeys]] internal property is ["ca", "nu"].

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

The initial value of the [[ocaleData]] internal property is implementation dependent within the constraints described in 9.1 and the following additional constraints:

- [[localeData]][locale] must have an hour12 property with a Boolean value for all locale values.

- [[localeData]][locale] must have a formats property for all locale values. The value of this property must be an array of objects, each of which has a subset of the properties shown in Table 3, where each property must have one of the values specified for the property in Table 3. Multiple objects in an array may use the same subset of the properties as long as they have different values for the properties. The following subsets must be available for each locale:
  - weekday, year, month, day, hour, minute, second
  - weekday, year, month, day
  - year, month, day
  - year, month
  - month, day
  - hour, minute, second
  - hour, minute

  Each of the objects must also have a pattern property, whose value is a String value that contains for each of the date and time format component properties of the object a substring starting with "{", followed by the name of the property, followed by "}".

EXAMPLE An implementation might include the following object as part of its English locale data: {hour: "numeric", minute: "2-digit", second: "2-digit", pattern: "{hour}:{minute}:{second}"}.

12.4 Properties of the DateTimeFormat Prototype Object

The DateTimeFormat prototype object is itself a DateTimeFormat object, whose internal properties are set as if it had been constructed using new DateTimeFormat().

In the following descriptions of functions that are properties of the DateTimeFormat prototype object, the phrase “this DateTimeFormat object” refers to the object that is the this value for the invocation of the function.

12.4.1 Globalization.DateTimeFormat.prototype.constructor

The initial value of Globalization.DateTimeFormat.prototype.constructor is the built-in DateTimeFormat constructor.

12.4.2 Globalization.DateTimeFormat.prototype.format ([date])

Returns a String value representing the result of calling ToNumber(date) according to the effective locale and the formatting options of this DateTimeFormat object.

1. If *date* is omitted let *y* be `Date.now()`; else let *y* be ToNumber(*date*).
2. If *y* is not a finite Number, then throw a **RangeError** exception.
3. Let *locale* be the value of the [[locale]] internal property of this DateTimeFormat object.
4. Let *nf* be the result of creating a new NumberFormat object as if by the expression `new Globalization.NumberFormat([locale])` where `Globalization.NumberFormat` is the standard built-in constructor with that name.
5. Let *nf2* be the result of creating a new NumberFormat object as if by the expression `new Globalization.NumberFormat([locale], {minimumIntegerDigits: 2})` where `Globalization.NumberFormat` is the standard built-in constructor with that name.
6. Let *tm* be the result of calling ToLocalTime with *y*, the value of the [[calendar]] internal property of this DateTimeFormat object, and the value of the [[timeZone]] internal property of this DateTimeFormat object.
7. Let *result* be the value of the [[pattern]] internal property of this DateTimeFormat object.
8. For each row of Table 3, except the header row, do:
   a. If this DateTimeFormat object has an internal property with the name given in the Property column of the row, then:
      i. Let *p* be the name given in the Property column of the row.
      ii. Let *f* be the value of the internal property named *p* of this DateTimeFormat object.
      iii. Let *v* be the value of the property named *p* of *tm*.
      iv. If *p* equals `"month"`, then increase *v* by 1.
      v. If *f* equals `"numeric"`, then let *fv* be the result of calling the format method of *nf* with argument *v*.

vi. Else if *f* equals **"2-digit"**, then
    1. Let *fv* be the result of calling the format method of *nf2* with argument *v*.
    2. If the length of *fv* is greater than 2, let *fv* be the result of calling the substring method of *fv* with *argument* length-2.

vii. Else if *f* equals **"narrow"**, **"short"**, or **"long"**, then let *fv* be an implementation, locale, and calendar dependent String value representing *f* in the desired form. If *p* equals **"month"**, then the String value may also depend on whether this DateTimeFormat object has a day property. If *p* equals **"timeZoneName"**, then the String value may also depend on the value of the isDST property of *tm*.

viii. Replace the substring of *result* that consists of **"{"**, *p*, and **"}"**, with *fv*.

9. Return *result*.

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

1. Apply calendrical calculations on date for the given calendar and time zone to produce weekday, era, year, month, day, hour, minute, second, and inDST values. The calculations should use best available information about the specified calendar and time zone. If the calendar is **"gregory"**, then the calculations are expected to match the algorithms specified in the ECMAScript Language Specification, 5.1 edition, 15.9.1, except that calculations are not bound by the restrictions on the use of best available information on time zones for local time zone adjustment and daylight saving time adjustment imposed by the ECMAScript Language Specification, 5.1 edition, 15.9.1.7 and 15.9.1.8.
2. Return an object with properties weekday, era, year, month, day, hour, minute, second, and inDST, each with the corresponding calculated value.

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

12.4.3 Globalization.DateTimeFormat.prototype.resolvedOptions

This named accessor property provides access to the locale and formatting options computed during initialization of the object. The value of the [[Get]] attribute is a function that returns a new object with properties locale, calendar, numberingSystem, timeZone, hour12, weekday, era, year, month, day, hour, minute, second, and timeZoneName, each with the value of the corresponding internal property of this DateTimeFormat object (see 12.5). The [[Set]] attribute is undefined.

NOTE In this version of the Globalization API, the timeZone property will remain undefined if no timeZone property was provided in the options object provided to the DateTimeFormat constructor. However, applications should not rely on this, as future versions may return a String value identifying the host environment's current time zone instead.

12.5 Properties of DateTimeFormat Instances

DateTimeFormat instances inherit properties from the DateTimeFormat prototype object. DateTimeFormat instances also have several internal properties 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 either the String value "UTC" or undefined.
- [[hour12]] is a Boolean value indicating whether 12-hour format (true) or 24-hour format (false) should be used.
- [[weekday]], [[era]], [[year]], [[month]], [[day]], [[hour]], [[minute]], [[second]], [[timeZoneName]] are each either undefined, indicating that the component is not used for formatting, or one of the String values given in Table 3, indicating how the component should be presented in the formatted output.
- [[pattern]] is a String value as described in 12.3.3.

13 Locale Sensitive Functions of the ECMAScript Language Specification

The ECMAScript Language Specification, edition 5.1 or successor, describes several locale sensitive functions. An ECMAScript implementations that implement this Globalization API shall implement these functions as described here.

13.1 Properties of the String Prototype Object

13.1.1 String.prototype.localeCompare (that [, localeList [, options]])

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

1. Let *collator* be the result of creating a new object as if by the expression **new** `Globalization.Collator(localeList, options)` where `Globalization.Collator` is the standard built-in constructor with that name.
2. Return the result of calling the compare method of *collator* with arguments **this** and *that*.

13.2 Properties of the Number Prototype Object

13.2.1 Number.prototype.toLocaleString ([localeList [, options]])

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

1. Let *numberFormat* be the result of creating a new object as if by the expression **new** `Globalization.NumberFormat(localeList, options)` where `Globalization.NumberFormat` is the standard built-in constructor with that name.
2. Return the result of calling the format method of *numberFormat* with argument **this**.

13.3 Properties of the Date Prototype Object

The properties of the Date prototype object described here use the abstract operation ToDateTimeOptions, specified in 12.2.1.

13.3.1 Date.prototype.toLocaleString ([localeList [, options]])

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

1. Let *options* be the result of calling the ToDateTimeOptions abstract operation with arguments *options*, **true**, and **true**.
2. Let *dateTimeFormat* be the result of creating a new object as if by the expression **new** `Globalization.DateTimeFormat(localeList, options)` where `Globalization.DateTimeFormat` is the standard built-in constructor with that name.
3. Return the result of calling the format method of *dateTimeFormat* with argument **this**.

13.3.2 Date.prototype.toLocaleDateString ([localeList [, options]])

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

1. Let *options* be the result of calling the ToDateTimeOptions abstract operation with arguments *options*, **true**, and **false**.

2. Let *dateFormat* be the result of creating a new object as if by the expression **`new Globalization.DateTimeFormat(localeList, options)`** where **`Globalization.DateTimeFormat`** is the standard built-in constructor with that name.
3. Return the result of calling the format method of *dateFormat* with argument **this**.

13.3.3 Date.prototype.toLocaleTimeString ([localeList [, options]])

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

1. Let *options* be the result of calling the ToDateTimeOptions abstract operation with arguments *options*, **false**, and **true**.
2. Let *timeFormat* be the result of creating a new object as if by the expression **`new Globalization.DateTimeFormat(localeList, options)`** where **`Globalization.DateTimeFormat`** is the standard built-in constructor with that name.
3. Return the result of calling the format method of *timeFormat* with argument **this**.