ecma INTERNATIONAL

Ecma/TC39/2012/041

Working Draft Standard ECMA-XXX

1st Edition / Draft 28 June 2012

ECMAScript Internationalization API Specification

Report errors and issues at https://bugs.ecmascript.org

Product: ECMAScript Internationalization API
Component: Specification Draft

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

Contents

Page

Introduction

The ECMAScript Internationalization API provides key language-sensitive functionality as a complement to the ECMAScript Language Specification, 5.1 edition or successor. Its functionality has been selected from that of well-established internationalization APIs such as those of the Internationalization Components for Unicode (ICU) library, of the .NET framework, or of the Java platform.

The API was developed by an ad-hoc group established by Ecma TC 39 in September 2010 based on a proposal by Nebojša Ćirić and Jungshik Shin (Google).

Internationalization of software is never complete. We expect significant enhancements in future editions of this specification.

ECMAScript Internationalization API Specification

1 Scope

This Standard defines the application programming interface for ECMAScript objects that support programs that need to adapt to the linguistic and cultural conventions used by different human languages and countries.

2 Conformance

A conforming implementation of the ECMAScript Internationalization 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 Internationalization API is permitted to provide additional objects, properties, and functions beyond those described in this specification. In particular, a conforming implementation of the ECMAScript Internationalization 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.

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

- The *options* property localeMatcher in all constructors and supportedLocalesOf methods.
- The *options* properties usage and sensitivity in the Collator constructor.
- The *options* properties style and currencyDisplay in the NumberFormat constructor.
- The *options* properties minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, and maximumSignificantDigits in the NumberFormat constructor, provided that the additional values are interpreted as integer values higher than the specified limits.
- The *options* property timeZone in the DateTimeFormat constructor, provided that the additional acceptable input values are case-insensitive matches of Zone or Link identifiers in the IANA time zone database and are canonicalized to Zone identifiers in the casing used in the database for the timeZone property of the object returned by DateTimeFormat.resolvedOptions, except that "Etc/GMT" shall be canonicalized to "UTC".
- The *options* properties listed in table 3 in the DateTimeFormat constructor.
- The *options* property formatMatcher in the DateTimeFormat constructor.

3 Normative References

The following referenced documents are required for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ECMA-262, ECMAScript Language Specification, 5.1 edition or successor
http://www.ecma-international.org/publications/standards/Ecma-262.htm

NOTE Throughout this document, the phrase “ES5, *x*”, where *x* is a sequence of numbers separated by periods, may be used as shorthand for “ECMAScript Language Specification, 5.1 edition, subclause *x*”.

ISO/IEC 10646:2003: Information Technology – Universal Multiple-Octet Coded Character Set (UCS) plus Amendment 1:2005 and Amendment 2:2006, plus additional amendments and corrigenda, or successor
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=39921

http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=40755
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=41419

ISO 4217:2008, Codes for the representation of currencies and funds, or successor
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=46121

IETF BCP 47:
- RFC 5646, Tags for Identifying Languages, or successor
  http://tools.ietf.org/html/rfc5646
- RFC 4647, Matching of Language Tags, or successor
  http://tools.ietf.org/html/rfc4647

IETF RFC 6067, BCP 47 Extension U, or successor
http://tools.ietf.org/html/rfc6067

IANA Time Zone Database
http://www.iana.org/time-zones/

The Unicode Standard, Version 5.0, or successor
http://www.unicode.org/versions/latest

Unicode Technical Standard 35, Unicode Locale Data Markup Language, version 2.0.1 or successor
http://unicode.org/reports/tr35/

4 Overview

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

4.1 Internationalization, Localization, and Globalization

Internationalization of software means designing it such that it supports or can be easily adapted to support the needs of users speaking different languages and having different cultural expectations, and enables worldwide communication between them. Localization then is the actual adaptation to a specific language and culture. Globalization of software is commonly understood to be the combination of internationalization and localization. Globalization starts at the lowest level by using a text representation that supports all languages in the world, and using standard identifiers to identify languages, countries, time zones, and other relevant parameters. It continues with using a user interface language and data presentation that the user understands, and finally often requires product-specific adaptations to the user's language, culture, and environment.

The ECMAScript 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 Internationalization API builds on this by providing a set of customizable language-sensitive functionality. The API is useful even for applications that themselves are not internationalized, as even applications targeting only one language and one region need to properly support that one language and region. However, the API also enables applications that support multiple languages and regions, even concurrently, as may be needed in server environments.

4.2 API Overview

The ECMAScript Internationalization 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.

The ECMAScript Internationalization 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 basic functionality (String.prototype.localeCompare, Number.prototype.toLocaleString, Date.prototype.toLocaleString,

Date.prototype.toLocaleDateString, and Date.prototype.toLocaleTimeString), it leaves the actual behavior of these functions largely up to implementations to define. The Internationalization API Specification provides additional functionality, control over the language and over details of the behavior to be used, and a more complete specification of required functionality.

Applications can use the API in two ways:

1. Directly, by using the constructors Intl.Collator, Intl.NumberFormat, or Intl.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.

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

4.3 Implementation Dependencies

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

- *The set of locales that an implementation supports with adequate localizations:* Linguists estimate the number of human languages to around 6000, and the more widely spoken ones have variations based on regions or other parameters. Even large locale data collections, such as the Common Locale Data Repository, cover only a subset of this large set. Implementations targeting resource-constrained devices may have to further reduce the subset.
- *The exact form of localizations such as format patterns:* In many cases locale-dependent conventions are not standardized, so different forms may exist side by side, or they vary over time. Different internationalization libraries may have implemented different forms, without any of them being actually wrong. In order to allow this API to be implemented on top of existing libraries, such variations have to be permitted.
- *Subsets of Unicode:* Some operations, such as collation, operate on strings that can include characters from the entire Unicode character set. However, both the Unicode standard and the ECMAScript standard allow implementations to limit their functionality to subsets of the Unicode character set. In addition, locale conventions typically don't specify the desired 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 ES5, 5.2.

- Internal properties, as described in ES5, 8.6.2.

- The List specification type, as described in ES5, 8.8.

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.

In addition, the Record specification type is used to describe data aggregations within the algorithms of this specification. A Record type value consists of one or more named fields. The value of each field is an ECMAScript type value. Field names are always enclosed in double brackets, for example [[field1]]. Field names can also be provided by a variable: The notation "[[<*name*>]]" denotes a field whose name is given by the variable *name*, which must have a String value. For example, if a variable *s* has the value **"a"**, then [[<*s*>]] denotes the field [[a]].

In specification text and algorithms, dot notation may be used to refer to a specific field of a Record value. For example, if *r* is a record, then *r*.[[field1]] is shorthand for "the field of *r* named [[field1]]".

For ECMAScript objects, this standard may use variable-named internal properties: The notation "[[<*name*>]]" denotes an internal property whose name is given by the variable *name*, which must have a String value. For example, if a variable *s* has the value **"a"**, then [[<*s*>]] denotes the [[a]] internal property.

6 Identification of Locales, Time Zones, and Currencies

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

6.1 Case Sensitivity and Case Mapping

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

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

6.2 Language Tags

The ECMAScript Internationalization 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.

BCP 47 language tags that meet those validity criteria of section 2.2.9 of RFC 5646 that can be verified without reference to the IANA Language Subtag Registry are considered structurally valid. All structurally valid language tags are valid for use with the APIs defined by this standard. However, the set of locales and thus language tags that an implementation supports with adequate localizations is implementation dependent. The constructors Collator, NumberFormat, and DateTimeFormat map the language tags used in requests to locales supported by their respective implementations.

6.2.1 Unicode Locale Extension Sequences

This standard uses the term "Unicode locale extension sequence" for any substring of a language tag that starts with a separator "-" and the singleton "u" and includes the maximum sequence of following non-singleton subtags and their preceding "-" separators.

6.2.2 IsStructurallyValidLanguageTag(locale)

The IsStructurallyValidLanguageTag abstract operation verifies that the *locale* argument (which must be a String value)

- represents a well-formed BCP 47 language tag as specified in RFC 5646 section 2.1, or successor,
- does not include duplicate variant subtags, and
- does not include duplicate singleton subtags.

The abstract operation returns true if *locale* can be generated from the ABNF grammar in section 2.1 of the RFC, starting with Language-Tag, and does not contain duplicate variant or singleton subtags (other than as a private use subtag). It returns false otherwise. Terminal value characters in the grammar are interpreted as the Unicode equivalents of the ASCII octet values given.

6.2.3 CanonicalizeLanguageTag (locale)

The CanonicalizeLanguageTag abstract operation returns the canonical and case-regularized form of the locale argument (which must be a String value that is a structurally valid BCP 47 language tag as verified by the IsStructurallyValidLanguageTag 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, but does not take the steps to bring a language tag into “extlang form” and to reorder variant subtags.

The specifications for extensions to BCP 47 language tags, such as RFC 6067, may include canonicalization rules for the extension subtag sequences they define that go beyond the canonicalization rules of RFC 5646 section 4.5. Implementations are allowed, but not required, to apply these additional rules.

6.2.4 DefaultLocale ()

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

6.3 Currency Codes

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

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

6.3.1 IsWellFormedCurrencyCode (currency)

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

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

6.4 Time Zone Names

The ECMAScript Internationalization API defines a single time zone name, **`"UTC"`**, which identifies the UTC time zone.

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

7 Requirements for Standard Built-in ECMAScript Objects

Unless specified otherwise in this document, the objects, functions, and constructors described in this standard are subject to the generic requirements and restrictions specified for standard built-in ECMAScript objects in the ECMAScript Language Specification 5.1 edition, introduction of clause 15, or successor.

8 The Intl Object

The Intl object is a standard built-in object that is the initial value of the `Intl` property of the global object.

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

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

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

8.1 Properties of the Intl Object

The value of each of the standard built-in properties of the Intl object is a constructor. The behavior of these constructors is specified in the following clauses: Collator (10), NumberFormat (11), and DateTimeFormat (12).

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 locales 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 Intl.Collator, Intl.NumberFormat, and Intl.DateTimeFormat have the following internal properties:

- [[availableLocales]] is a List that contains structurally valid (6.2.2) and canonicalized (6.2.3) BCP 47 language tags identifying the locales for which the implementation provides the functionality of the constructed objects. Language tags on the list must not have a Unicode locale extension sequence. The list must include the value returned by the DefaultLocale abstract operation (6.2.4), and must not include duplicates. Implementations must include in [[availableLocales]] locales that can serve as fallbacks in the algorithm used to resolve locales (see 9.2.5). For example, implementations that provide a "de-DE" locale must include a "de" locale that can serve as a fallback for requests such as "de-AT" and "de-CH". For locales that in current usage would include a script subtag (such as Chinese locales), old-style language tags without script subtags must be included such that, for example, requests for "zh-TW" and "zh-HK" lead to output in traditional Chinese rather than the default simplified Chinese. The ordering of the locales within [[availableLocales]] is irrelevant.
- [[relevantExtensionKeys]] is 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 Intl.Collator) and [[localeData]] (for Intl.NumberFormat and Intl.DateTimeFormat) are objects that have properties for each locale contained in [[availableLocales]]. The value of each of these properties must be an object that 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 12.2.3) 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"]}}.

9.2 Abstract Operations

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

9.2.1 CanonicalizeLocaleList (locales)

The abstract operation CanonicalizeLocaleList takes the following steps:

1. If *locales* is **undefined**, then
    a. Return a new empty List.
2. Let *seen* be a new empty List.
3. Let *O* be ToObject(*locales*).
4. Let *lenValue* be the result of calling the [[Get]] internal method of *O* with the argument **`"length"`**.
5. Let *len* be ToUint32(*lenValue*).
6. Let *k* be 0.
7. Repeat, while *k* < *len*
    a. Let *Pk* be ToString(*k*).
    b. Let *kPresent* be the result of calling the [[HasProperty]] internal method of *O* with argument *Pk*.
    c. If *kPresent* is **true**, then
        i. Let *kValue* be the result of calling the [[Get]] internal method of *O* with argument *Pk*.
        ii. If the type of *kValue* is not String or Object, then throw a **TypeError** exception.
        iii. Let *tag* be ToString(*kValue*).
        iv. If the result of calling the abstract operation IsStructurallyValidLanguageTag (defined in 6.2.2), passing *tag* as the argument, is **false**, then throw a **RangeError** exception.
        v. Let *tag* be the result of calling the abstract operation CanonicalizeLanguageTag (defined in 6.2.3), passing *tag* as the argument.
        vi. If *tag* is not an element of *seen*, then append *tag* as the last element of *seen*.
    d. Increase *k* by 1.
8. Return *seen*.

NOTE Non-normative summary: The abstract operation interprets the *locales* argument as an array and copies its elements into a List, validating the elements as structurally valid language tags and canonicalizing them, and omitting duplicates.

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

9.2.2 BestAvailableLocale (availableLocales, locale)

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

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

9.2.3 LookupMatcher (availableLocales, requestedLocales)

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

1. Let *i* be 0.
2. Let *len* be the number of elements in *requestedLocales*.
3. Let *availableLocale* be **undefined**.
4. Repeat while *i* < *len* and *availableLocale* is **undefined**:
    a. Let *locale* be the element of *requestedLocales* at 0-origined list position *i*.
    b. Let *noExtensionsLocale* be the String value that is *locale* with all Unicode locale extension sequences removed.
    c. Let *availableLocale* be the result of calling the BestAvailableLocale abstract operation (defined in 9.2.2) with arguments *availableLocales* and *noExtensionsLocale*.
    d. Increase *i* by 1.
5. Let *result* be a new Record.
6. If *availableLocale* is not **undefined**, then
    a. Set *result*.[[locale]] to *availableLocale*.
    b. If *locale* and *noExtensionsLocale* are not the same String value, then
        i. Let *extension* be the String value consisting of the first substring of *locale* that is a Unicode locale extension sequence.
        ii. Let *extensionIndex* be the character position of the initial `"-"` of the first Unicode locale extension sequence within *locale*.
        iii. Set *result*.[[extension]] to *extension*.
        iv. Set *result*.[[extensionIndex]] to *extensionIndex*.
7. Else
    a. Set *result*.[[locale]] to the value returned by the DefaultLocale abstract operation (defined in 6.2.4).
8. Return *result*.

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

9.2.4 BestFitMatcher (availableLocales, requestedLocales)

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

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

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

The following steps are taken:

1. Let *matcher* be the value of *options*.[[localeMatcher]].
2. If *matcher* is `"lookup"` then
    a. Let *r* be the result of calling the LookupMatcher abstract operation (defined in 9.2.3) with arguments *availableLocales* and *requestedLocales*.
3. Else
    a. Let *r* be the result of calling the BestFitMatcher abstract operation (defined in 9.2.4) with arguments *availableLocales* and *requestedLocales*.
4. Let *foundLocale* be the value of *r*.[[locale]].
5. If *r* has an [[extension]] field, then
    a. Let *extension* be the value of *r*.[[extension]].
    b. Let *extensionIndex* be the value of *r*.[[extensionIndex]].
    c. Let *split* be the standard built-in function object defined in ES5, 15.5.4.14.
    d. Let *extensionSubtags* be the result of calling the [[Call]] internal method of *split* with *extension* as the **this** value and an argument list containing the single item `"-"`.
    e. Let *extensionSubtagsLength* be the result of calling the [[Get]] internal method of *extensionSubtags* with argument `"length"`.
6. Let *result* be a new Record.
7. Set *result*.[[dataLocale]] to *foundLocale*.
8. Let *supportedExtension* be `"-u"`.
9. Let *i* be 0.
10. Let *len* be the result of calling the [[Get]] internal method of *relevantExtensionKeys* with argument `"length"`.
11. Repeat while *i* < *len*:
    a. Let *key* be the result of calling the [[Get]] internal method of *relevantExtensionKeys* with argument ToString(*i*).
    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 result of calling the [[Get]] internal method of *keyLocaleData* with argument `"0"`.
    e. Let *supportedExtensionAddition* be `""`.
    f. Let *indexOf* be the standard built-in function object defined in ES5, 15.4.4.14.
    g. If *extensionSubtags* is not **undefined**, then
        i. Let *keyPos* be the result of calling the [[Call]] internal method of *indexOf* with *extensionSubtags* as the **this** value and an argument list containing the single item *key*.
        ii. If *keyPos* ≠ -1 then
            1. If *keyPos* + 1 < *extensionSubtagsLength* and the length of the result of calling the [[Get]] internal method of *extensionSubtags* with argument ToString(*keyPos* +1) is greater than 2, then
                a. Let *requestedValue* be the result of calling the [[Get]] internal method of *extensionSubtags* with argument ToString(*keyPos* + 1).
                b. Let *valuePos* be the result of calling the [[Call]] internal method of *indexOf* with *keyLocaleData* as the **this** value and an argument list containing the single item *requestedValue*.
                c. If *valuePos* ≠ -1, then
                    i. Let *value* be *requestedValue*.
                    ii. Let *supportedExtensionAddition* be the concatenation of `"-"`, *key*, `"-"`, and *value*.
            2. Else
                a. Let *valuePos* be the result of calling the [[Call]] internal method of *indexOf* with *keyLocaleData* as the **this** value and an argument list containing the single item `"true"`.
                b. If *valuePos* ≠ -1, then
                    i. Let *value* be `"true"`.
    h. If *options* has a field [[<*key*>]] then
        i. Let *optionsValue* be the value of *options*.[[<*key*>]].
        ii. If the result of calling the [[Call]] internal method of *indexOf* with *keyLocaleData* as the **this** value and an argument list containing the single item *optionsValue* is not -1, then

1. If *optionsValue* is not equal to *value*, then
    a. Let *value* be *optionsValue*.
    b. Let *supportedExtensionAddition* be "".

i. Set *result*.[[<*key*>]] to *value*.
j. Append *supportedExtensionAddition* to *supportedExtension*.
k. Increase *i* by 1.

12. If the length of *supportedExtension* is greater than 2, then
    a. Let *preExtension* be the substring of *foundLocale* from position 0 to position *extensionIndex* -1.
    b. Let *postExtension* be the substring of *foundLocale* from position *extensionIndex* to the end of the string.
    c. Let *foundLocale* be the concatenation of *preExtension*, *supportedExtension*, and *postExtension*.
13. Set *result*.[[locale]] to *foundLocale*.
14. Return *result*.

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

9.2.6 LookupSupportedLocales (availableLocales, requestedLocales)

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

1. Let *len* be the number of elements in *requestedLocales*.
2. Let *subset* be a new empty List.
3. Let *k* be 0.
4. Repeat while *k* < *len*
    a. Let *locale* be the element of *requestedLocales* at 0-origined list position *k*.
    b. Let *noExtensionsLocale* be the String value that is *locale* with all Unicode locale extension sequences removed.
    c. Let *availableLocale* be the result of calling the BestAvailableLocale abstract operation (defined in 9.2.2) with arguments *availableLocales* and *noExtensionsLocale*.
    d. If *availableLocale* is not **undefined**, then append *locale* to the end of *subset*.
    e. Increment *k* by 1.
5. Let *subsetArray* be a new Array object whose elements are the same values in the same order as the elements of *subset*.
6. Return *subsetArray*.

9.2.7 BestFitSupportedLocales (availableLocales, requestedLocales)

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

9.2.8 SupportedLocales (availableLocales, requestedLocales, options)

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

1. If *options* is not **undefined**, then
    a. Let *options* be ToObject(*options*).

   b. Let *matcher* be the result of calling the [[Get]] internal method of *options* with argument `"localeMatcher"`.
   c. If *matcher* is not **undefined**, then
      i. Let *matcher* be ToString(*matcher*).
      ii. If *matcher* is not `"lookup"` or `"best fit"`, then throw a **RangeError** exception.
2. If *matcher* is **undefined** or `"best fit"` then
   a. Let *subset* be the result of calling the BestFitSupportedLocales abstract operation (defined in 9.2.7) with arguments *availableLocales* and *requestedLocales*.
3. Else
   a. Let *subset* be the result of calling the LookupSupportedLocales abstract operation (defined in 9.2.6) with arguments *availableLocales* and *requestedLocales*.
4. Let *freeze* be the standard built-in function object defined in ES5, 15.2.3.9.
5. Call the [[Call]] internal method of *freeze* with an argument list containing the single item *subset*.
6. Set the [[Extensible]] internal property of *subset* to **true**.
7. Return *subset*.

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

The GetOption abstract operation extracts a the value of the property named *property* from the provided *options* object, converts it to the required *type*, checks whether it is one of a List of allowed *values*, and fills in a *fallback* value if necessary.

1. Let *value* be the result of calling the [[Get]] internal method of *options* with argument *property*.
2. If *value* is neither **undefined** nor **null** then
   a. If *type* is `"boolean"` then let *value* be ToBoolean(*value*).
   b. If *type* is `"string"` then let *value* be ToString(*value*).
   c. If *type* is `"number"` then let *value* be ToNumber(*value*).
   d. If *values* is not **undefined**, then
      i. If *values* does not contain an element equal to *value*, then throw a **RangeError** exception.
   e. Return *value*.
3. Else return *fallback*.

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

The GetNumberOption abstract operation 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.

1. Let *value* be the result of calling the [[Get]] internal method of *options* with argument *property*.
2. If *value* is neither **undefined** nor **null** then
   a. Let *value* be ToNumber(*value*).
   b. If *value* is **NaN** or less than *minimum* or greater than *maximum*, throw a **RangeError** exception.
   c. Return floor(*value*).
3. Else return *fallback*.

10 Collator Objects

10.1 The Intl.Collator Constructor

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

10.1.1 Initializing an Object as a Collator

10.1.1.1 InitializeCollator (collator, locales, options)

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

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

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

Key	Property	Type	Values
kn	numeric	"boolean"	
kk	normalization	"boolean"	
kf	caseFirst	"string"	"upper", "lower", "false"

The following steps are taken:

1. If *collator* has an [[initializedIntlObject]] internal property with value **true**, throw a **TypeError** exception.
2. Set the [[initializedIntlObject]] internal property of *collator* to **true**.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. If *options* is **undefined**, then
    a. 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.
5. Else
    a. Let *options* be ToObject(*options*).
6. Let *u* be the result of calling the GetOption abstract operation (defined in 9.2.9) with arguments *options*, `"usage"`, `"string"`, a List containing the two String values `"sort"` and `"search"`, and `"sort"`.
7. Set the [[usage]] internal property of *collator* to *u*.
8. Let *Collator* be the standard built-in object that is the initial value of Intl.Collator.
9. If *u* is `"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*.
10. Let *opt* be a new Record.
11. Let *matcher* be the result of calling the GetOption abstract operation with arguments *options*, `"localeMatcher"`, `"string"`, a List containing the two String values `"lookup"` and `"best fit"`, and `"best fit"`.
12. Set *opt*.[[localeMatcher]] to *matcher*.
13. For each row in Table 1, except the header row, do:
    a. Let *key* be the name given in the Key column of the row.
    b. Let *value* be the result of calling the GetOption abstract operation, passing as arguments *options*, the name given in the Property column of the row, the string given in the Type column of the row, a List containing the Strings given in the Values column of the row or **undefined** if no strings are given, and **undefined**.
    c. If the string given in the Type column of the row is `"boolean"` and *value* is not **undefined**, then
        i. Let *value* be ToString(*value*).
    d. Set *opt*.[[<*key*>]] to *value*.
14. Let *relevantExtensionKeys* be the value of the [[relevantExtensionKeys]] internal property of *Collator*.
15. Let *r* be the result of calling the ResolveLocale abstract operation (defined in 9.2.5) with the [[availableLocales]] internal property of *Collator*, *requestedLocales*, *opt*, *relevantExtensionKeys*, and *localeData*.
16. Set the [[locale]] internal property of *collator* to the value of *r*.[[locale]].
17. Let *i* be 0.
18. Let *len* be the result of calling the [[Get]] internal method of *relevantExtensionKeys* with argument `"length"`.
19. Repeat while *i* < *len*:
    a. Let *key* be the result of calling the [[Get]] internal method of *relevantExtensionKeys* with argument ToString(*i*).
    b. If *key* is `"co"`, then
        i. Let *property* be `"collation"`.
        ii. Let *value* be the value of *r*.[[co]].
        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 *r*.[[<*key*>]].

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 [[<*property*>]] internal property of *collator* to *value*.

e. Increase *i* by 1.

20. Let *s* be the result of calling the GetOption abstract operation with arguments *options*, **"sensitivity"**, **"string"**, a List containing the four String values **"base"**, **"accent"**, **"case"**, and **"variant"**, and **undefined**.
21. If *s* is **undefined**, then
    a. If *u* is **"sort"**, then let s be **"variant"**.
    b. Else
        i. Let *dataLocale* be the value of *r*.[[dataLocale]].
        ii. Let *dataLocaleData* be the result of calling the [[Get]] internal operation of *localeData* with argument *dataLocale*.
        iii. Let *s* be the result of calling the [[Get]] internal operation of *dataLocaleData* with argument **"sensitivity"**.
22. Set the [[sensitivity]] internal property of *collator* to *s*.
23. Let *ip* be the result of calling the GetOption abstract operation with arguments *options*, **"ignorePunctuation"**, **"boolean"**, **undefined**, and **false**.
24. Set the [[ignorePunctuation]] internal property of *collator* to *ip*.
25. Set the [[boundCompare]] internal property of *collator* to **undefined**.
26. Set the [[initializedCollator]] internal property of *collator* to **true**.

10.1.2 The Intl.Collator Constructor Called as a Function

10.1.2.1 Intl.Collator.call (this [, locales [, options]])

When **Intl.Collator** is called as a function rather than as a constructor, it accepts the optional arguments *locales* and *options* and takes the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. If **this** is the standard built-in Intl object defined in 8 or **undefined**, then
    a. Return the result of creating a new object as if by the expression new **Intl.Collator(locales, options)**, where **Intl.Collator** is the standard built-in constructor defined in 10.1.3.
4. Let *obj* be the result of calling ToObject passing the **this** value as the argument.
5. If the [[Extensible]] internal property of *obj* is **false**, throw a **TypeError** exception.
6. Call the InitializeCollator abstract operation (defined in 10.1.1.1) with arguments *obj*, *locales*, and *options*.
7. Return *obj*.

10.1.3 The Intl.Collator Constructor Used in a new Expression

10.1.3.1 new Intl.Collator ([locales [, options]])

When **Intl.Collator** is called as part of a **new** expression, it is a constructor: it initializes the newly created object.

The [[Prototype]] internal property of the newly constructed object is set to the original Intl.Collator prototype object, the one that is the initial value of **Intl.Collator.prototype** (10.2.1).

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

Collator-specific properties of the newly constructed object are set using the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. Call the InitializeCollator abstract operation (defined in 10.1.1.1), passing as arguments the newly constructed object, *locales*, and *options*.

10.2 Properties of the Intl.Collator Constructor

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

10.2.1 Intl.Collator.prototype

The value of **`Intl.Collator.prototype`** is the built-in Intl.Collator prototype object (10.3).

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

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

When the **`supportedLocalesOf`** method of Intl.Collator is called, the following steps are taken:

1. If *options* is not provided, then let *options* be **undefined**.
2. Let *availableLocales* be the value of the [[availableLocales]] internal property of the standard built-in object that is the initial value of Intl.Collator.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. Return the result of calling the SupportedLocales abstract operation (defined in 9.2.8) with arguments *availableLocales*, *requestedLocales*, and *options*.

10.2.3 Internal Properties

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

The value of the [[relevantExtensionKeys]] internal property is an array that must include the element **`"co"`**, may include any or all of the elements **`"kn"`**, **`"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 case level and the strength through the sensitivity property of the options object. The "co" key in the language tag is supported only for collator specializations, and the keys "kb", "kh", and "vt" are not allowed in this version of the Internationalization API. Support for the remaining keys is implementation dependent.

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

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

10.3 Properties of the Intl.Collator Prototype Object

The Intl.Collator prototype object is itself an Intl.Collator instance as specified in 10.4, whose internal properties are set as if it had been constructed by the expression **`Intl.Collator.call({})`** with the standard built-in values of Intl.Collator and Function.prototype.call.

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

exception is thrown if the this value is not an object or an object that does not have an [[initializedCollator]] internal property with value **true**.

10.3.1 Intl.Collator.prototype.constructor

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

10.3.2 Intl.Collator.prototype.compare

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

The value of the [[Get]] attribute is a function that takes the following steps:

1. If the [[boundCompare]] internal property of this Collator object is **undefined**, then:
    a. Let *F* be a Function object, with internal properties and the length property set as specified for built-in functions in ES5, 15, or successor, that takes the arguments *x* and *y* and performs the following steps:
        i. If *x* is not provided, then let *x* be **undefined**.
        ii. If *y* is not provided, then let *y* be **undefined**.
        iii. Let *X* be ToString(*x*).
        iv. Let *Y* be ToString(*y*).
        v. Return the result of calling the CompareStrings abstract operation (defined below) with arguments **this**, *X*, and *Y*.
    b. Let *bind* be the standard built-in function object defined in ES5, 15.3.4.5.
    c. Let *bc* be the result of calling the [[Call]] internal method of *bind* with *F* as the **this** value and an argument List containing the single item **this**.
    d. Set the [[boundCompare]] internal property of this Collator object to *bc*.
2. Return the value of the [[boundCompare]] internal property of this Collator object.

NOTE The function returned by [[Get]] is bound to this Collator object so that it can be passed directly to Array.prototype.sort or other functions.

The value of the [[Set]] attribute is **undefined**.

When the CompareStrings abstract operation is called with arguments *collator* (which must be a Collator object), *x* and *y* (which must be String values), it returns a Number other than **NaN** that represents the result of a locale-sensitive String comparison of *x* with *y*. The two Strings are compared in an implementation-defined fashion. The result is intended to order String values in the sort order specified by the effective locale and collation options computed during construction of *collator*, and will be negative, zero, or positive, depending on whether *x* comes before *y* in the sort order, the Strings are equal under the sort order, or *x* comes after *y* in the sort order, respectively.

The sensitivity of *collator* is interpreted as follows:

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

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

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

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

The CompareStrings abstract operation with any given *collator* argument, if considered as a function of the remaining two arguments *x* and *y*, must be a consistent comparison function (as defined in ES5, 15.4.4.11) 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 It is recommended that the CompareStrings abstract operation be implemented following Unicode Technical Standard 10, Unicode Collation Algorithm (available at http://unicode.org/reports/tr10/), using tailorings for the effective locale and collation options of *collator*. It is recommended that implementations use the tailorings provided by the Common Locale Data Repository (available at http://cldr.unicode.org/).

NOTE 2 Applications should not assume that the behavior of the CompareStrings abstract operation for Collator instances with the same resolved options will remain the same for different versions of the same implementation.

10.3.3 Intl.Collator.prototype.resolvedOptions ()

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

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal property of this Collator object (see 10.4): locale, usage, sensitivity, ignorePunctuation, collation, as well as those properties shown in Table 1 whose keys are included in the [[relevantExtensionKeys]] internal property of the standard built-in object that is the initial value of Intl.Collator.

10.4 Properties of Intl.Collator Instances

Intl.Collator instances inherit properties from the Intl.Collator prototype object. Their [[Class]] internal property value is **"Object"**.

Intl.Collator instances and other objects that have been successfully initialized as a Collator have [[initializedIntlObject]] and [[initializedCollator]] internal properties whose values are **true**.

Objects that have been successfully initialized as a Collator 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.

Objects that have been successfully initialized as a Collator 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 Intl.Collator:

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

Finally, objects that have been successfully initialized as a Collator have a [[boundCompare]] internal property that caches the function returned by the compare accessor (10.3.2).

11 NumberFormat Objects

11.1 The Intl.NumberFormat Constructor

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

11.1.1 Initializing an Object as a NumberFormat

11.1.1.1 InitializeNumberFormat (numberFormat, locales, options)

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

The following steps are taken:

1. If *numberFormat* has an [[initializedIntlObject]] internal property with value **true**, throw a **TypeError** exception.
2. Set the [[initializedIntlObject]] internal property of *numberFormat* to **true**.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. If *options* is **undefined**, then
    a. 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.
5. Else
    a. Let *options* be ToObject(*options*).
6. Let *opt* be a new Record.
7. Let *matcher* be the result of calling the GetOption abstract operation (defined in 9.2.9) with the arguments *options*, `"localeMatcher"`, `"string"`, a List containing the two String values `"lookup"` and `"best fit"`, and `"best fit"`.
8. Set *opt*.[[localeMatcher]] to *matcher*.
9. Let *NumberFormat* be the standard built-in object that is the initial value of Intl.NumberFormat.
10. Let *localeData* be the value of the [[localeData]] internal property of *NumberFormat*.
11. Let *r* be the result of calling the ResolveLocale abstract operation (defined in 9.2.5) with the [[availableLocales]] internal property of *NumberFormat*, *requestedLocales*, *opt*, the [[relevantExtensionKeys]] internal property of *NumberFormat*, and *localeData*.
12. Set the [[locale]] internal property of *numberFormat* to the value of *r*.[[locale]].
13. Set the [[numberingSystem]] internal property of *numberFormat* to the value of *r*.[[nu]].
14. Let *dataLocale* be the value of *r*.[[dataLocale]].
15. Let *s* be the result of calling the GetOption abstract operation with the arguments *options*, `"style"`, `"string"`, a List containing the three String values `"decimal"`, `"percent"`, and `"currency"`, and `"decimal"`.
16. Set the [[style]] internal property of *numberFormat* to *s*.
17. Let *c* be the result of calling the GetOption abstract operation with the arguments *options*, `"currency"`, `"string"`, **undefined**, and **undefined**.
18. If *c* is not undefined and the result of calling the IsWellFormedCurrencyCode abstract operation (defined in 6.3.1) with argument *c* is **false**, then throw a **RangeError** exception.
19. If *s* is `"currency"` and *c* is **undefined**, throw a **TypeError** exception.
20. If *s* is `"currency"` then
    a. Let *c* be the result of converting *c* to upper case as specified in 6.1.
    b. Set the [[currency]] internal property of *numberFormat* to *c*.
    c. Let *cDigits* be the result of calling the CurrencyDigits abstract operation (defined below) with argument *c*.
21. Let *cd* be the result of calling the GetOption abstract operation with the arguments *options*, `"currencyDisplay"`, `"string"`, a List containing the three String values `"code"`, `"symbol"`, and `"name"`, and `"symbol"`.
22. If *s* is `"currency"` then set the [[currencyDisplay]] internal property of *numberFormat* to *cd*.
23. Let *mnid* be the result of calling the GetNumberOption abstract operation (defined in 9.2.10) with arguments *options*, `"minimumIntegerDigits"`, 1, 21, and 1.
24. Set the [[minimumIntegerDigits]] internal property of *numberFormat* to *mnid*.
25. If *s* is `"currency"` then let *mnfdDefault* be *cDigits*; else let *mnfdDefault* be 0.

26. Let *mnfd* be the result of calling the GetNumberOption abstract operation with arguments *options*, **"minimumFractionDigits"**, 0, 20, and *mnfdDefault*.
27. Set the [[minimumFractionDigits]] internal property of *numberFormat* to *mnfd*.
28. If *s* is **"currency"** then let *mxfdDefault* be max(*mnfd*, *cDigits*); else if *s* is **"percent"** then let *mxfdDefault* be max(*mnfd*, 0); else let *mxfdDefault* be max(*mnfd*, 3).
29. Let *mxfd* be the result of calling the GetNumberOption abstract operation with arguments *options*, **"maximumFractionDigits"**, *mnfd*, 20, and *mxfdDefault*.
30. Set the [[maximumFractionDigits]] internal property of *numberFormat* to *mxfd*.
31. Let *mnsd* be the result of calling the [[Get]] internal method of *options* with argument **"minimumSignificantDigits"**.
32. Let *mxsd* be the result of calling the [[Get]] internal method of *options* with argument **"maximumSignificantDigits"**.
33. If *mnsd* is not **undefined** or *mxsd* is not **undefined**, then:
    a. Let *mnsd* be the result of calling the GetNumberOption abstract operation with arguments *options*, **"minimumSignificantDigits"**, 1, 21, and 1.
    b. Let *mxsd* be the result of calling the GetNumberOption abstract operation with arguments *options*, **"maximumSignificantDigits"**, *mnsd*, 21, and 21.
    c. Set the [[minimumSignificantDigits]] internal property of *numberFormat* to *mnsd*, and the [[maximumSignificantDigits]] internal property of *numberFormat* to *mxsd*.
34. Let *g* be the result of calling the GetOption abstract operation with the arguments *options*, **"useGrouping"**, **"boolean"**, **undefined**, and **true**.
35. Set the [[useGrouping]] internal property of *numberFormat* to *g*.
36. Let *dataLocaleData* be the result of calling the [[Get]] internal method of *localeData* with argument *dataLocale*.
37. Let *patterns* be the result of calling the [[Get]] internal method of *dataLocaleData* with argument **"patterns"**.
38. Assert: *patterns* is an object (see 11.2.3).
39. Let *stylePatterns* be the result of calling the [[Get]] internal method of *patterns* with argument *s*.
40. Set the [[positivePattern]] internal property of *numberFormat* to the result of calling the [[Get]] internal method of *stylePatterns* with the argument **"positivePattern"**.
41. Set the [[negativePattern]] internal property of *numberFormat* to the result of calling the [[Get]] internal method of *stylePatterns* with the argument **"negativePattern"**.
42. Set the [[boundFormat]] internal property of *numberFormat* to **undefined**.
43. Set the [[initializedNumberFormat]] internal property of *numberFormat* to **true**.

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

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

11.1.2 The Intl.NumberFormat Constructor Called as a Function

11.1.2.1 Intl.NumberFormat.call (this [, locales [, options]])

When **Intl.NumberFormat** is called as a function rather than as a constructor, it accepts the optional arguments *locales* and *options* and takes the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. If **this** is the standard built-in Intl object defined in 8 or **undefined**, then
    a. Return the result of creating a new object as if by the expression **new Intl.NumberFormat(locales, options)**, where **Intl.NumberFormat** is the standard built-in constructor defined in 11.1.3.
4. Let *obj* be the result of calling ToObject passing the **this** value as the argument.
5. If the [[Extensible]] internal property of *obj* is **false**, throw a **TypeError** exception.
6. Call the InitializeNumberFormat abstract operation (defined in 11.1.1.1) with arguments *obj*, *locales*, and *options*.
7. Return *obj*.

11.1.3 The Intl.NumberFormat Constructor Used in a new Expression

11.1.3.1 new Intl.NumberFormat ([locales [, options]])

When **`Intl.NumberFormat`** is called as part of a **`new`** expression, it is a constructor: it initializes the newly created object.

The [[Prototype]] internal property of the newly constructed object is set to the original Intl.NumberFormat prototype object, the one that is the initial value of **`Intl.NumberFormat.prototype`** (11.2.1).

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

NumberFormat-specific properties of the newly constructed object are set using the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. Call the InitializeNumberFormat abstract operation (defined in 11.1.1.1), passing as arguments the newly constructed object, *locales*, and *options*.

11.2 Properties of the Intl.NumberFormat Constructor

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

11.2.1 Intl.NumberFormat.prototype

The value of **`Intl.NumberFormat.prototype`** is the built-in Intl.NumberFormat prototype object (11.3).

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

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

When the **`supportedLocalesOf`** method of Intl.NumberFormat is called, the following steps are taken:

1. If *options* is not provided, then let *options* be **undefined**.
2. Let *availableLocales* be the value of the [[availableLocales]] internal property of the standard built-in object that is the initial value of Intl.NumberFormat.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. Return the result of calling the SupportedLocales abstract operation (defined in 9.2.8) with arguments *availableLocales*, *requestedLocales*, and *options*.

11.2.3 Internal Properties

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

The 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. Intl.NumberFormat, however, requires that the currency of a currency format is specified through the currency property in the options objects.

The value of the [[localeData]] internal property is implementation defined 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 have properties with the names of the three number format styles: **`"decimal"`**, **`"percent"`**, and **`"currency"`**. Each of these properties in turn must be an object with the properties

positivePattern and negativePattern. The value of these properties must be string values that contain a substring `"{number}"`; the values within the currency property must also contain a substring `"{currency}"`. The pattern strings must not contain any characters in the General Category "Number, decimal digit" as specified by the Unicode Standard.

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

11.3 Properties of the Intl.NumberFormat Prototype Object

The Intl.NumberFormat prototype object is itself an Intl.NumberFormat instance as specified in 11.4, whose internal properties are set as if it had been constructed by the expression `Intl.NumberFormat.call({})` with the standard built-in values of Intl.NumberFormat and Function.prototype.call.

In the following descriptions of functions that are properties of the Intl.NumberFormat prototype object, the phrase "this NumberFormat object" refers to the object that is the this value for the invocation of the function; a **TypeError** exception is thrown if the this value is not an object or an object that does not have an [[initializedNumberFormat]] internal property with value **true**.

11.3.1 Intl.NumberFormat.prototype.constructor

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

11.3.2 Intl.NumberFormat.prototype.format

This named accessor property returns a function that formats a number according to the effective locale and the formatting options of this NumberFormat object.

The value of the [[Get]] attribute is a function that takes the following steps:

1. If the [[boundFormat]] internal property of this NumberFormat object is **undefined**, then:
    a. Let *F* be a Function object, with internal properties and the length property set as specified for built-in functions in ES5, 15, or successor, that takes the argument *value* and performs the following steps:
        i. If *value* is not provided, then let *value* be **undefined**.
        ii. Let *x* be ToNumber(*value*).
        iii. Return the result of calling the FormatNumber abstract operation (defined below) with arguments **this** and *x*.
    b. Let *bind* be the standard built-in function object defined in ES5, 15.3.4.5.
    c. Let *bf* be the result of calling the [[Call]] internal method of *bind* with *F* as the **this** value and an argument list containing the single item **this**.
    d. Set the [[boundFormat]] internal property of this NumberFormat object to *bf*.
2. Return the value of the [[boundFormat]] internal property of this NumberFormat object.

NOTE The function returned by [[Get]] is bound to this NumberFormat object so that it can be passed directly to Array.prototype.map or other functions.

The value of the [[Set]] attribute is **undefined**.

When the FormatNumber abstract operation is called with arguments *numberFormat* (which must be a NumberFormat object) and *x* (which must be a Number value), it returns a String value representing *x* according to the effective locale and the formatting options of *numberFormat*.

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

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

The following steps are taken:

1. Let *negative* be **false**.
2. 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**.
3. 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 *numberFormat* is **"percent"**, let *x* be 100 × *x*.
    c. If the [[minimumSignificantDigits]] and [[maximumSignificantDigits]] internal properties of *numberFormat* are present, then
        i. Let *n* be the result of calling the ToRawPrecision abstract operation (defined below), passing as arguments *x* and the values of the [[minimumSignificantDigits]] and [[maximumSignificantDigits]] internal properties of *numberFormat*.
    d. Else
        i. Let *n* be the result of calling the ToRawFixed abstract operation (defined below), passing as arguments *x* and the values of the [[minimumIntegerDigits]], [[minimumFractionDigits]], and [[maximumFractionDigits]] internal properties of *numberFormat*.
    e. If the value of the [[numberingSystem]] internal property of *numberFormat* matches one of the values in the "Numbering System" column of Table 2 below, then
        i. Let *digits* be an array whose 10 String valued elements are the UTF-16 string representations of 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 *numberFormat* is **true**, then insert an ILND String representing a grouping separator into an ILND set of locations within the integer part of *n*.
4. If *negative* is **true**, then let *result* be the value of the [[negativePattern]] internal property of *numberFormat*; else let *result* be the value of the [[positivePattern]] internal property of *numberFormat*.
5. Replace the substring **"{number}"** within *result* with *n*.
6. If the value of the [[style]] internal property of *numberFormat* is **"currency"**, then:
    a. Let *currency* be the value of the [[currency]] internal property of *numberFormat*.
    b. If the value of the [[currencyDisplay]] internal property of *numberFormat* is **"code"**, then let *cd* be *currency*.
    c. Else if the value of the [[currencyDisplay]] internal property of *numberFormat* 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 *numberFormat* 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*.
7. Return *result*.

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

1. Let *p* be *maxPrecision*.
2. If *x* = 0, then
    a. Let *m* be the String consisting of *p* occurrences of the character **"0"**.
    b. Let *e* be 0.

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

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

1. Let *f* be *maxFraction*.
2. Let *n* be an integer for which the exact mathematical value of $n \div 10^f - x$ is as close to zero as possible. If there are two such *n*, pick the larger *n*.
3. If $n = 0$, let *m* be the String `"0"`. Otherwise, let *m* be the String consisting of the digits of the decimal representation of *n* (in order, with no leading zeroes).
4. If $f \ne 0$, then
    a. Let *k* be the number of characters in *m*.
    b. If $k \le f$, 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

11.3.3 Intl.NumberFormat.prototype.resolvedOptions ()

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

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal property of this NumberFormat object (see 11.4): locale, numberingSystem, style, currency, currencyDisplay, minimumIntegerDigits, minimumFractionDigits, maximumFractionDigits, minimumSignificantDigits, maximumSignificantDigits, and useGrouping. Properties whose corresponding internal properties are not present are not assigned.

11.4 Properties of Intl.NumberFormat Instances

Intl.NumberFormat instances inherit properties from the Intl.NumberFormat prototype object. Their [[Class]] internal property value is `"Object"`.

Intl.NumberFormat instances and other objects that have been successfully initialized as a NumberFormat have [[initializedIntlObject]] and [[initializedNumberFormat]] internal properties whose values are **true**.

Objects that have been successfully initialized as a NumberFormat 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. It is only present when [[style]] has the value **"currency"**.
- [[currencyDisplay]] is one of the String values **"code"**, **"symbol"**, or **"name"**, specifying whether to display the currency as an ISO 4217 alphabetic currency code, a localized currency symbol, or a localized currency name if formatting with the **"currency"** style. It is only present when [[style]] has the value **"currency"**.
- [[minimumIntegerDigits]] is a non-negative integer Number value indicating the minimum integer digits to be used. Numbers will be padded with leading zeroes if necessary.
- [[minimumFractionDigits]] and [[maximumFractionDigits]] are non-negative integer Number values indicating the minimum and maximum fraction digits to be used. Numbers will be rounded or padded with trailing zeroes if necessary.
- [[minimumSignificantDigits]] and [[maximumSignificantDigits]] are positive integer Number values indicating the minimum and maximum fraction digits to be shown. Either none or both of these properties are present; if they are, they override minimum and maximum integer and fraction digits – the formatter uses however many integer and fraction digits are required to display the specified number of significant digits.
- [[useGrouping]] is a Boolean value indicating whether a grouping separator should be used.
- [[positivePattern]] and [[negativePattern]] are String values as described in 11.2.3.

Finally, objects that have been successfully initialized as a NumberFormat have a [[boundFormat]] internal property that caches the function returned by the format accessor (11.3.2).

12 DateTimeFormat Objects

12.1 The Intl.DateTimeFormat Constructor

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

12.1.1 Initializing an Object as a DateTimeFormat

12.1.1.1 InitializeDateTimeFormat (dateTimeFormat, locales, options)

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

Several DateTimeFormat 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 *dateTimeFormat* has an [[initializedIntlObject]] internal property with value **true**, throw a **TypeError** exception.
2. Set the [[initializedIntlObject]] internal property of *dateTimeFormat* to **true**.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. Let *options* be the result of calling the ToDateTimeOptions abstract operation (defined below) with arguments *options*, **"any"**, and **"date"**.
5. Let *opt* be a new Record.
6. Let *matcher* be the result of calling the GetOption abstract operation (defined in 9.2.9) with arguments *options*, **"localeMatcher"**, **"string"**, a List containing the two String values **"lookup"** and **"best fit"**, and **"best fit"**.
7. Set *opt*.[[localeMatcher]] to *matcher*.
8. Let *DateTimeFormat* be the standard built-in object that is the initial value of Intl.DateTimeFormat.
9. Let *localeData* be the value of the [[localeData]] internal property of *DateTimeFormat*.
10. Let *r* be the result of calling the ResolveLocale abstract operation (defined in 9.2.5) with the [[availableLocales]] internal property of *DateTimeFormat*, *requestedLocales*, *opt*, the [[relevantExtensionKeys]] internal property of *DateTimeFormat*, and *localeData*.
11. Set the [[locale]] internal property of *dateTimeFormat* to the value of *r*.[[locale]].
12. Set the [[calendar]] internal property of *dateTimeFormat* to the value of *r*.[[ca]].
13. Set the [[numberingSystem]] internal property of *dateTimeFormat* to the value of *r*.[[nu]].
14. Let *dataLocale* be the value of *r*.[[dataLocale]].
15. Let *tz* be the result of calling the [[Get]] internal method of *options* with argument **"timeZone"**.
16. 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 **"UTC"**, then throw a **RangeError** exception.
17. Set the [[timeZone]] internal property of *dateTimeFormat* to *tz*.
18. Let *hr12* be the result of calling the GetOption abstract operation with arguments *options*, **"hour12"**, **"boolean"**, **undefined**, and **undefined**.
19. Let *dataLocaleData* be the result of calling the [[Get]] internal method of *localeData* with argument *dataLocale*.
20. If *hr12* is **undefined**, then let *hr12* be the result of calling the [[Get]] internal method of *dataLocaleData* with argument **"hour12"**.
21. Set the [[hour12]] internal property of *dateTimeFormat* to *hr12*.
22. Let *opt* be a new Record.
23. For each row of Table 3, except the header row, do:
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *value* be the result of calling the GetOption abstract operation, passing as argument *options*, the name given in the Property column of the row, **"string"**, a List containing the strings given in the Values column of the row, and **undefined**.
    c. Set *opt*.[[<*prop*>]] to *value*.
24. Let *formats* be the result of calling the [[Get]] internal method of *dataLocaleData* with argument **"formats"**.
25. Let *matcher* be the result of calling the GetOption abstract operation with arguments *options*, **"formatMatcher"**, **"string"**, a List containing the two String values **"basic"** and **"best fit"**, and **"best fit"**.
26. If *matcher* is **"basic"** then
    a. Let *bestFormat* be the result of calling the BasicFormatMatcher abstract operation (defined below) with *opt* and *formats*.
27. Else
    a. Let *bestFormat* be the result of calling the BestFitFormatMatcher abstract operation (defined below) with *opt* and *formats*.
28. For each row in Table 3, except the header row, do
    a. Let *prop* be the name given in the Property column of the row.
    b. Let *pDesc* be the result of calling the [[GetOwnProperty]] internal method of *bestFormat* with argument *prop*.
    c. If *pDesc* is not **undefined**, then
        i. Let *p* be the result of calling the [[Get]] internal method of *bestFormat* with argument *prop*.
        ii. Set the [[<*prop*>]] internal property of *dateTimeFormat* to *p*.
29. Let *pattern* be the result of calling the [[Get]] internal method of *bestFormat* with argument **"pattern"**.
30. Set the [[pattern]] internal property of *dateTimeFormat* to *pattern*.

31. Set the [[boundFormat]] internal property of *dateTimeFormat* to **undefined**.
32. Set the [[initializedDateTimeFormat]] internal property of *dateTimeFormat* to **true**.

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

1. If *options* is **undefined**, then let *options* be **null**, else let *options* be ToObject(*options*).
2. Let *create* be the standard built-in function object defined in ES5, 15.2.3.5.
3. Let *options* be the result of calling the [[Call]] internal method of *create* with **undefined** as the **this** value and an argument list containing the single item *options*.
4. Let *needDefaults* be **true**.
5. If *required* is **"date"** or **"any"**, 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 *needDefaults* be **false**.
6. If *required* is **"time"** or **"any"**, 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 *needDefaults* be **false**.
7. If *needDefaults* is **true** and *defaults* is either **"date"** or **"all"**, then
    a. For each of the property names **"year"**, **"month"**, **"day"**:
        i. Call the [[DefineOwnProperty]] internal method of *options* with the property name, Property Descriptor {[[Value]]: **"numeric"**, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
8. If *needDefaults* is **true** and *defaults* is either **"time"** or **"all"**, then
    a. For each of the property names **"hour"**, **"minute"**, **"second"**:
        i. Call the [[DefineOwnProperty]] internal method of *options* with the property name, Property Descriptor {[[Value]]: **"numeric"**, [[Writable]]: **true**, [[Enumerable]]: **true**, [[Configurable]]: **true**}, and **false**.
9. Return *options*.

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

1. Let *removalPenalty* be 120.
2. Let *additionPenalty* be 20.
3. Let *longLessPenalty* be 8.
4. Let *longMorePenalty* be 6.
5. Let *shortLessPenalty* be 6.
6. Let *shortMorePenalty* be 3.
7. Let *bestScore* be -**Infinity**.
8. Let *bestFormat* be **undefined**.
9. Let *i* be 0.
10. Let *len* be the result of calling the [[Get]] internal method of *formats* with argument "**length**".
11. Repeat while *i* < *len*:
    a. Let *format* be the result of calling the [[Get]] internal method of *formats* with argument ToString(*i*).
    b. Let *score* be 0.
    c. For each *property* shown in Table 3:
        i. Let *optionsProp* be *options*.[[<*property*>]].
        ii. Let *formatPropDesc* be the result of calling the [[GetOwnProperty]] internal method of *format* with argument *property*.
        iii. If *formatPropDesc* is not **undefined**, then
            1. Let *formatProp* be the result of calling the [[Get]] internal method of *format* with argument *property*.
        iv. If *optionsProp* is **undefined** and *formatProp* is not **undefined**, then decrease *score* by *additionPenalty*.
        v. Else if *optionsProp* is not **undefined** and *formatProp* is **undefined**, then decrease *score* by *removalPenalty*.

vi. 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* = 2, decrease *score* by *longMorePenalty*.
6. Else if *delta* = 1, decrease *score* by *shortMorePenalty*.
7. Else if *delta* = -1, decrease *score* by *shortLessPenalty*.
8. Else if *delta* = -2, decrease *score* by *longLessPenalty*.

d. If *score* > *bestScore*, then
i. Let *bestScore* be *score*.
ii. Let *bestFormat* be *format*.

e. Increase *i* by 1.

12. Return *bestFormat*.

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

12.1.2 The Intl.DateTimeFormat Constructor Called as a Function

12.1.2.1 Intl.DateTimeFormat.call (this [, locales [, options]])

When `Intl.DateTimeFormat` is called as a function rather than as a constructor, it accepts the optional arguments *locales* and *options* and takes the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. If **this** is the standard built-in Intl object defined in 8 or **undefined**, then
   a. Return the result of creating a new object as if by the expression new `Intl.DateTimeFormat(locales, options)`, where `Intl.DateTimeFormat` is the standard built-in constructor defined in 12.1.3.
4. Let *obj* be the result of calling ToObject passing the **this** value as the argument.
5. If the [[Extensible]] internal property of *obj* is **false**, throw a **TypeError** exception.
6. Call the InitializeDateTimeFormat abstract operation (defined in 12.1.1.1) with arguments *obj*, *locales*, and *options*.
7. Return *obj*.

12.1.3 The Intl.DateTimeFormat Constructor Used in a new Expression

12.1.3.1 new Intl.DateTimeFormat ([locales [, options]])

When `Intl.DateTimeFormat` is called as part of a `new` expression, it is a constructor: it initializes the newly created object.

The [[Prototype]] internal property of the newly constructed object is set to the original Intl.DateTimeFormat prototype object, the one that is the initial value of `Intl.DateTimeFormat.prototype` (12.2.1).

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

DateTimeFormat-specific properties of the newly constructed object are set using the following steps:

1. If *locales* is not provided, then let *locales* be **undefined**.
2. If *options* is not provided, then let *options* be **undefined**.
3. Call the InitializeDateTimeFormat abstract operation (defined in 12.1.1.1), passing as arguments the newly constructed object, *locales*, and *options*.

12.2 Properties of the Intl.DateTimeFormat Constructor

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

12.2.1 Intl.DateTimeFormat.prototype

The value of **`Intl.DateTimeFormat.prototype`** is the built-in Intl.DateTimeFormat prototype object (12.3).

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

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

When the **`supportedLocalesOf`** method of Intl.DateTimeFormat is called, the following steps are taken:

1. If *options* is not provided, then let *options* be **undefined**.
2. Let *availableLocales* be the value of the [[availableLocales]] internal property of the standard built-in object that is the initial value of Intl.DateTimeFormat.
3. Let *requestedLocales* be the result of calling the CanonicalizeLocaleList abstract operation (defined in 9.2.1) with argument *locales*.
4. Return the result of calling the SupportedLocales abstract operation (defined in 9.2.8) with arguments *availableLocales*, *requestedLocales*, and *options*.

12.2.3 Internal Properties

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

The 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 value of the [[localeData]] internal property is implementation defined 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}"}.

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

12.3 Properties of the Intl.DateTimeFormat Prototype Object

The Intl.DateTimeFormat prototype object is itself an Intl.DateTimeFormat instance as specified in 12.4, whose internal properties are set as if it had been constructed by the expression `Intl.DateTimeFormat.call({})` with the standard built-in values of Intl.DateTimeFormat and Function.prototype.call.

In the following descriptions of functions that are properties of the Intl.DateTimeFormat prototype object, the phrase "this DateTimeFormat object" refers to the object that is the this value for the invocation of the function; a **TypeError** exception is thrown if the this value is not an object or an object that does not have an [[initializedDateTimeFormat]] internal property with value **true**.

12.3.1 Intl.DateTimeFormat.prototype.constructor

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

12.3.2 Intl.DateTimeFormat.prototype.format

This named accessor property returns a function that formats a date according to the effective locale and the formatting options of this DateTimeFormat object.

The value of the [[Get]] attribute is a function that takes the following steps:

1. If the [[boundFormat]] internal property of this DateTimeFormat object is **undefined**, then:
    a. Let *F* be a Function object, with internal properties and the length property set as specified for built-in functions in ES5, 15, or successor, that takes the argument *date* and performs the following steps:
        i. If *date* is not provided or is **undefined**, then let *x* be the result as if by the expression `Date.now()` where `Date.now` is the standard built-in function defined in ES5, 15.9.4.4.
        ii. Else let *x* be ToNumber(*date*).
        iii. Return the result of calling the FormatDateTime abstract operation (defined below) with arguments **this** and *x*.
    b. Let *bind* be the standard built-in function object defined in ES5, 15.3.4.5.
    c. Let *bf* be the result of calling the [[Call]] internal method of *bind* with *F* as the **this** value and an argument list containing the single item **this**.
    d. Set the [[boundFormat]] internal property of this DateTimeFormat object to *bf*.
2. Return the value of the [[boundFormat]] internal property of this DateTimeFormat object.

NOTE The function returned by [[Get]] is bound to this DateTimeFormat object so that it can be passed directly to Array.prototype.map or other functions.

The value of the [[Set]] attribute is **undefined**.

When the FormatDateTime abstract operation is called with arguments *dateTimeFormat* (which must be a DateTimeFormat object) and *x* (which must be a Number value), it returns a String value representing *x* (interpreted as a time value as specified in ES5, 15.9.1.1) according to the effective locale and the formatting options of *dateTimeFormat*.

1. If *x* is not a finite Number, then throw a **RangeError** exception.
2. Let *locale* be the value of the [[locale]] internal property of *dateTimeFormat*.
3. Let *nf* be the result of creating a new NumberFormat object as if by the expression `new Intl.NumberFormat([locale], {useGrouping: false})` where `Intl.NumberFormat` is the standard built-in constructor defined in 11.1.3.
4. Let *nf2* be the result of creating a new NumberFormat object as if by the expression `new Intl.NumberFormat([locale], {minimumIntegerDigits: 2, useGrouping: false})` where `Intl.NumberFormat` is the standard built-in constructor defined in 11.1.3.

5. Let *tm* be the result of calling the ToLocalTime abstract operation (defined below) with *x*, the value of the [[calendar]] internal property of *dateTimeFormat*, and the value of the [[timeZone]] internal property of *dateTimeFormat*.
6. Let *result* be the value of the [[pattern]] internal property of *dateTimeFormat*.
7. For each row of Table 3, except the header row, do:
    a. If *dateTimeFormat* 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 [[<*p*>]] internal property of *dateTimeFormat*.
        iii. Let *v* be the value of *tm*.[[<*p*>]].
        iv. If *p* is **"month"**, then increase *v* by 1.
        v. If *f* is **"numeric"**, then
            1. Let *fv* be the result of calling the FormatNumber abstract operation (defined in 11.3.2) with arguments *nf* and *v*.
        vi. Else if *f* is **"2-digit"**, then
            1. Let *fv* be the result of calling the FormatNumber abstract operation with arguments *nf2* and *v*.
            2. If the length of *fv* is greater than 2, let *fv* be the substring of *fv* containing the last two characters.
        vii. Else if *f* is **"narrow"**, **"short"**, or **"long"**, then let *fv* be a String value representing *f* in the desired form; the String value depends upon the implementation and the effective locale and calendar of *dateTimeFormat*. If *p* is **"month"**, then the String value may also depend on whether *dateTimeFormat* has a [[day]] internal property. If *p* is **"timeZoneName"**, then the String value may also depend on the value of the [[isDST]] field of *tm*.
        viii. Replace the substring of *result* that consists of **"{"**, *p*, and **"}"**, with *fv*.
8. Return *result*.

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

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 must match the algorithms specified in ES5, 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 ES5, 15.9.1.7 and 15.9.1.8.
2. Return a Record with fields [[weekday]], [[era]], [[year]], [[month]], [[day]], [[hour]], [[minute]], [[second]], and [[inDST]], each with the corresponding calculated value.

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

12.3.3 Intl.DateTimeFormat.prototype.resolvedOptions ()

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

The function returns a new object whose properties and attributes are set as if constructed by an object literal assigning to each of the following properties the value of the corresponding internal property of this DateTimeFormat object (see 12.4): locale, calendar, numberingSystem, timeZone, hour12, weekday, era, year, month, day, hour, minute, second, and timeZoneName. Properties whose corresponding internal properties are not present are not assigned.

NOTE In this version of the ECMAScript Internationalization API, the timeZone property will remain undefined if no timeZone property was provided in the options object provided to the Intl.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.

12.4 Properties of Intl.DateTimeFormat Instances

Intl.DateTimeFormat instances inherit properties from the Intl.DateTimeFormat prototype object. Their [[Class]] internal property value is `"Object"`.

Intl.DateTimeFormat instances and other objects that have been successfully initialized as a DateTimeFormat have [[initializedIntlObject]] and [[initializedDateTimeFormat]] internal properties whose values are **true**.

Objects that have been successfully initialized as a DateTimeFormat 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 absent, 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 12.2.3.

Finally, objects that have been successfully initialized as a DateTimeFormat have a [[boundFormat]] internal property that caches the function returned by the format accessor (12.3.2).

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 implementation that implements this Internationalization API shall implement these functions as described here.

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

13.1 Properties of the String Prototype Object

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

This definition supersedes the definition provided in ES5, 15.5.4.9.

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

1. Call CheckObjectCoercible passing the **this** value as its argument.
2. Let *S* be the result of calling ToString passing the **this** value as its argument.
3. Let *That* be ToString(*that*).
4. If *locales* is not provided, then let *locales* be **undefined**.
5. If *options* is not provided, then let *options* be **undefined**.
6. Let *collator* be the result of creating an object as if by the expression `new Intl.Collator(locales, options)` where `Intl.Collator` is the standard built-in constructor defined in 10.1.3.
7. Return the result of calling the CompareStrings abstract operation (defined in 10.3.2) with arguments *collator*, *S*, and *That*.

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

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

13.2 Properties of the Number Prototype Object

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

This definition supersedes the definition provided in ES5, 15.7.4.3.

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

1. Let *x* be this Number value (as defined in ES5, 15.7.4).
2. If *locales* is not provided, then let *locales* be **undefined**.
3. If *options* is not provided, then let *options* be **undefined**.
4. Let *numberFormat* be the result of creating a new object as if by the expression **`new Intl.NumberFormat(locales, options)`** where **`Intl.NumberFormat`** is the standard built-in constructor defined in 11.1.3.
5. Return the result of calling the FormatNumber abstract operation (defined in 11.3.2) with arguments *numberFormat* and *x*.

13.3 Properties of the Date Prototype Object

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

This definition supersedes the definition provided in ES5, 15.9.5.5.

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

1. Let *x* be this time value (as defined in ES5, 15.9.5).
2. If *x* is **NaN**, then return **`"Invalid Date"`**.
3. If *locales* is not provided, then let *locales* be **undefined**.
4. If *options* is not provided, then let *options* be **undefined**.
5. Let *options* be the result of calling the ToDateTimeOptions abstract operation (defined in 12.1.1) with arguments *options*, **`"any"`**, and **`"all"`**.
6. Let *dateTimeFormat* be the result of creating a new object as if by the expression **`new Intl.DateTimeFormat(locales, options)`** where **`Intl.DateTimeFormat`** is the standard built-in constructor defined in 12.1.3.
7. Return the result of calling the FormatDateTime abstract operation (defined in 12.3.2) with arguments *dateTimeFormat* and *x*.

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

This definition supersedes the definition provided in ES5, 15.9.5.6.

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

1. Let *x* be this time value (as defined in ES5, 15.9.5).
2. If *x* is **NaN**, then return **`"Invalid Date"`**.
3. If *locales* is not provided, then let *locales* be **undefined**.
4. If *options* is not provided, then let *options* be **undefined**.
5. Let *options* be the result of calling the ToDateTimeOptions abstract operation (defined in 12.1.1) with arguments *options*, **`"date"`**, and **`"date"`**.

6. Let *dateFormat* be the result of creating a new object as if by the expression **`new Intl.DateTimeFormat(locales, options)`** where **`Intl.DateTimeFormat`** is the standard built-in constructor defined in 12.1.3.
7. Return the result of calling the FormatDateTime abstract operation (defined in 12.3.2) with arguments *dateTimeFormat* and *x*.

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

This definition supersedes the definition provided in ES5, 15.9.5.7.

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

1. Let *x* be this time value (as defined in ES5, 15.9.5).
2. If x is **NaN**, then return **`"Invalid Date"`**.
3. If *locales* is not provided, then let *locales* be **undefined**.
4. If *options* is not provided, then let *options* be **undefined**.
5. Let *options* be the result of calling the ToDateTimeOptions abstract operation (defined in 12.1.1) with arguments *options*, **`"time"`**, and **`"time"`**.
6. Let *timeFormat* be the result of creating a new object as if by the expression **`new Intl.DateTimeFormat(locales, options)`** where **`Intl.DateTimeFormat`** is the standard built-in constructor defined in 12.1.3.
7. Return the result of calling the FormatDateTime abstract operation (defined in 12.3.2) with arguments *dateTimeFormat* and *x*.