Intl.DurationFormat Design Doc

[public viewable design doc]

Status: DRAFT
Author: ftang@chromium.org
Last Updated: 2022-9-9

Objective

Implement the Intl.DurationFormat proposal in v8 JavaScript engine.

Background

Intl.DurationFormat proposal is now a Stage 3 proposal part of the ECMA-402 ECMAScript Internationalization API program. ICU4C library already partially implements related functionality in C++ and this project implements the necessary adopting code in V8 to bind the JavaScript API to the C++ implementation.

Overview

We will follow the footsteps of Intl.Locale , Intl.RelativeTimeFormat and Intl.ListFormat, Intl.DisplyNames and Intl.Segmenter.

Detailed Design

High Level Design

We will define a flag "**Intl.DurationFormat**" (--harmony_intl_duration_format) to enable this in progress feature. Such flag will later be referenced by C++ code as **FLAG_harmony_intl_duration_format**.

We intend to introduce one new class **JSDurationFormat**

- **JSDurationFormat** is needed to track the necessary information for Intl.DurationFormat, including resolved **locale**, **style** and **fields**, and ICU object to be used.

JSDurationFormat class will be implemented in the following files:

- v8/src/objects/js-duration-format.h
- v8/src/objects/js-duration-format.cc

- v8/src/objects/js-duration-format-inl.h
- v8/src/objects/js-duration-format.tq

We will also hold an intl_duration_format_function in the src/objects/contexts.h .

Public Observable Functions

We need to implement the following five public observable functions in v8. These function are implemented as static method in the JSDurationFormat class:

- Intl.DurationFormat ([locales [, options]]) : The constructor.
    - As static MaybeHandle<JSDurationFormat> New(Isolate* isolate, Handle<Map> map, Handle<Object> locales, Handle<Object> options);
- Intl.DurationFormat.supportedLocalesOf (locales [, options]) : class function to expose the supporting locales.
    - As static const std::set<std::string>& GetAvailableLocales();
- Intl.DurationFormat.prototype.format (duration) : member function to format the duration to a String.
    - As static MaybeHandle<String> Format(Isolate* isolate, Handle<JSDurationFormat> df, Handle<Object> duration);
- Intl.DurationFormat.prototype.formatToParts (duration) : member function to format the duration to an Array of Objects- one for each part.
    - As static MaybeHandle<JSArray> FormatToParts(Isolate* isolate, Handle<JSDurationFormat> df, Handle<Object> duration);
- Intl.DurationFormat.prototype.resolvedOptions ( ) : member function to return an Object to expose the resolved settings.
    - As static Handle<JSObject> ResolvedOptions(Isolate* isolate, Handle<JSDurationFormat> format_holder);

Enum Classes

Style is an enum class defined in JSDurationFormat. This enum is defined to encode the [[Style]] internal slot:

```
enum class Style {
    kLong,
    kShort,
    kNarrow,
    kDigital,
  };
```

This information needs a total of 2 bits in the internal slot.

Display is an enum class defined in JSDurationFormat. This enum is defined to encode the [[YearsDisplay]], [[MonthsDisplay]], [[WeeksDisplay]], [[DaysDisplay]], [[HoursDisplay]], [[MinutesDisplay]], [[SecondsDisplay]], [[MillisecondsDisplay]], [[MicrosecondsDisplay]], [[NanosecondsDisplay]] internal slot:

```
enum class Display {
    kAuto,
    kAlways
  };
```

Each internal slot needs 1 bit to store the value, resulting total 10 bits storage requirement.

FieldStyle is an enum class defined in JSDurationFormat. This enum is defined to encode the [[YearsStyle]], [[MonthsStyle]], [[WeeksStyle]], [[DaysStyle]], [[HoursStyle]], [[MinutesDisplay]], [[SecondsStyle]], [[MillisecondsStyle]], [[MicrosecondsStyle]], [[NanosecondsStyle]] internal slot:

```
  enum class FieldStyle {
    kLong,
    kShort,
    kNarrow,
    kNumeric,
    k2Digit,
    kUndefined
  };
```

While there are 6 possible values in the definition. When stored into internal slots, only the first five are valid values. And for the first four fields, only the first three values are valid. For the last three fields, only the first four values are valid. Therefore, we need a total 4 x 2 + 3 x 3 + 3 x 2 = 23 bits in the internal slot to store this information.

Internal Storage

Since we need 2 bits for [[Style]], 10 bits for [[xxxDisplay]], 23 bits for [[xxxStyle]] and also another four bits for the fractional digits, we cannot store these information into one 32 bits field but need two. We declare a style_flags and a display_flags field as below:

```

```

```
type JSDurationFormatStyle extends int32 constexpr
'JSDurationFormat::Style';
type JSDurationFormatFieldStyle extends int32
constexpr 'JSDurationFormat::FieldStyle';
type JSDurationFormatDisplay extends int32
constexpr 'JSDurationFormat::Display';
bitfield struct JSDurationFormatStyleFlags extends uint31 {
  style: JSDurationFormatStyle: 2 bit;
  years_style: JSDurationFormatFieldStyle: 2 bit;
  months_style: JSDurationFormatFieldStyle: 2 bit;
  weeks_style: JSDurationFormatFieldStyle: 2 bit;
  days_style: JSDurationFormatFieldStyle: 2 bit;
  hours_style: JSDurationFormatFieldStyle: 3 bit;
  minutes_style: JSDurationFormatFieldStyle: 3 bit;
  seconds_style: JSDurationFormatFieldStyle: 3 bit;
  milliseconds_style: JSDurationFormatFieldStyle: 2 bit;
  microseconds_style: JSDurationFormatFieldStyle: 2 bit;
  nanoseconds_style: JSDurationFormatFieldStyle: 2 bit;
}
bitfield struct JSDurationFormatDisplayFlags extends uint31 {
  years_display: JSDurationFormatDisplay: 1 bit;
  months_display: JSDurationFormatDisplay: 1 bit;
  weeks_display: JSDurationFormatDisplay: 1 bit;
  days_display: JSDurationFormatDisplay: 1 bit;
  hours_display: JSDurationFormatDisplay: 1 bit;
  minutes_display: JSDurationFormatDisplay: 1 bit;
  seconds_display: JSDurationFormatDisplay: 1 bit;
  milliseconds_display: JSDurationFormatDisplay: 1 bit;
  microseconds_display: JSDurationFormatDisplay: 1 bit;
  nanoseconds_display: JSDurationFormatDisplay: 1 bit;
  fractional_digits: int32: 4 bit;
}
```

- style_flags only use 25 out of total 32 bits, with 7 bits unused.
- display_flags only use 14 bits out of total 32 bits, with 18 bits unused.

```
extern class JSDurationFormat extends JSObject {
  style_flags: SmiTagged<JSDurationFormatStyleFlags>;
  display_flags: SmiTagged<JSDurationFormatDisplayFlags>;
  icu_locale: Foreign;  // Managed<icu::Locale>
  icu_number_formatter:
```

```
      Foreign;  // Managed<icu::number::LocalizedNumberFormatter>
}
```

We also need two Foreign fields to store an icu::Locale and an icu::number::LocalizedNumberFormatter for other settings. These information will be used to construct several icu::number::LocalizedNumberFormatter and icu::ListFormatter during the format/formatToParts operation.

New Heap Symbols

The following symbols needed to be added in src/init/heap-symbols.h

- “daysDisplay”
- “digital”
- “fractionalDigits”
- “hoursDisplay”
- “microsecondsDisplay”
- “millisecondsDisplay”
- “minutesDisplay”
- “monthsDisplay”
- “secondsDisplay”
- “2-digit”
- “weeksDisplay”
- “yearsDisplay”

Required Changes in Other Classes

Required Changes in JSNumberFormat

To better reuse the code, we expose an internal function inside JSNumberFormat to be called:

```
  static const icu::UnicodeString NumberingSystemFromSkeleton(
      const icu::UnicodeString& skeleton);
```

This function extracts the numberingSystem from a skeleton, both in the type of icu::UnicodeString.

Required Changes in Temporal

Two Abstract Operations defined in the Intl.DurationFormat proposal, ToDurationRecord and IsValidDurationRecord are the same (or intended to be the same) as Abstract Operations defined in the Temporal proposal (see ToTemporalPartialDurationRecord and IsValidDuration). For better code reuse, we expose the following two pre-existing functions from src/objects/js-temporal-object.cc to its header file under the temporal namespace:

```
// For Intl.DurationFormat

// #sec-temporal-time-duration-records
struct TimeDurationRecord {
  double days;
  double hours;
  double minutes;
  double seconds;
  double milliseconds;
  double microseconds;
  double nanoseconds;

  // #sec-temporal-createtimedurationrecord
  static Maybe<TimeDurationRecord> Create(Isolate* isolate, double days,
                                          double hours, double minutes,
                                          double seconds, double
milliseconds,
                                          double microseconds,
                                          double nanoseconds);
};

// #sec-temporal-duration-records
// Cannot reuse DateDurationRecord here due to duplicate days.
struct DurationRecord {
  double years;
  double months;
  double weeks;
  TimeDurationRecord time_duration;
  // #sec-temporal-createdurationrecord
  static Maybe<DurationRecord> Create(Isolate* isolate, double years,
                                      double months, double weeks, double
days,
                                      double hours, double minutes,
```

```
                                      double seconds, double milliseconds,
                                      double microseconds, double
nanoseconds);
};

// #sec-temporal-topartialduration
Maybe<DurationRecord> ToPartialDuration(
    Isolate* isolate, Handle<Object> temporal_duration_like_obj,
    const DurationRecord& input);

// #sec-temporal-isvalidduration
bool IsValidDuration(Isolate* isolate, const DurationRecord& dur);
```

Files To Be Added/Changed

Note: mainly by following code example of Implementation of Intl.Locale in V8

Basic Facility

- src/flag-definitions.h
  - Use V(harmony_intl_duration_format, "Intl.DurationFormat") to define FLAG_harmony_intl_duration_format
- BUILD.gn : List the new source files
- src/objects/contexts.h
  - V(INTL_DURATION_FORMAT_FUNCTION_INDEX, JSFunction, intl_duration_format_function)
- include/v8.h
  - Constant for UseCounter

The Usage of JSDurationFormat

- **src/builtins/builtins-definitions.h** Define the following function by CPP() macro
  - CPP(DurationFormatConstructor)
  - CPP(DurationFormatPrototypeFormat)
  - CPP(DurationFormatPrototypeFormatToParts)
  - CPP(DurationFormatPrototypeResolvedOptions)
  - CPP(DurationFormatSupportedLocalesOf)

- src/builtins/builtins-intl.cc :
  - Implement the above functions by BUILTIN() macro

Testing Code

See the "Testing Plan" section for more detail.

- v8/test/intl/duration/*
    - Unit test in JS
- https://github.com/tc39/test262/tree/main/test/intl402/DurationFormat

Open Issues (OI)

This section tracked all the open issues we discovered during the design. Once the issue is resolved, it will be moved into the Closed Issues (OI) section below with the resolution.

- OI1- Is MeasureFormat the right ICU class to use to implement this
    - MeasureFormat::formatMeasures
    - LocalizedNumberFormatter + ...
- OI2- How to implement formatToParts- Do we need to add a new API to ICU to support that?
    - Proposing the following to ICU TC for enhancement to MeasureFormatOI3- the singular/plural issue of fields option
    - Discussing in issues/44
- ~~OI4 - How to control numberingSystem~~
- OI5- How to determine which locale is supported? Locale is supported by
    - A. Intl.NumberFormat AND Intl.ListFormat AND Intl.PluralRules
    - **B. Intl.NumberFormat AND Intl.ListFormat**
    - C. Intl.NumberFormat
    - D. ???
- OI6 - order of getting the field not the same as Temporal

Closed Issues (CI)

This sections tracked all the resolved open issues:

- CI4 - How to control numberingSystem
    - The MessageFormat constructor will take the NumberFormat

Project Information

- **Design Doc (this doc):** This document
- **Tracking Bug**: v8/issues/detail?id=11660
- **Chrome Platform Status dashboard**: 5193021205774336
- **Public Intend To (Prototype, Trial, Ship) messages**
    - **I2P**: KwmaQ7LqxKY/m/ruIY6Kg6CAAJ

    - **I2T**:
    - **I2S**:
- **Changes**
    - https://chromium-review.googlesource.com/c/v8/v8/+/3833436
    - Obsoleted prototype
        - ~~https://chromium-review.googlesource.com/c/v8/v8/+/2762664~~
        - ~~https://chromium-review.googlesource.com/c/v8/v8/+/2776518~~
        - ~~https://chromium-review.googlesource.com/c/v8/v8/+/2775300~~
- **TL**: Frank Tang ftang@google.com
- **TLM**: Nebojša Ćirić cira@google.com
- **SWE/Developer**: Frank Tang ftang@google.com
- **Reviewers**:
    - **Jakob Kummerow** jkummerow@chromium.org
    - **Shu-yu Guo** syg@chromium.org

- **Discussion Hangout**: Intl-v8 group hangout

Testing Plan

How to Run JavaScript Unit Tests

```
$ tools/dev/gm.py x64.release
$ tools/run-tests.py  --outdir=out/x64.release \
        test262/*Duration*  intl/*duration*
```

How to Run C++ Unit Tests

Remember to rebuild cctest before you run the run-test.py

```
$ tools/dev/gm.py x64.release
$ tools/run-tests.py  --outdir=out/x64.release cctest/*
```

Location

- We should submit new tests for this function into test262/tree/master/test/intl402/DurationFormat.

Work Estimates

Work Breakdown

Target Schedule

Reference Documents & Site

- About the Specification
  - Intl.DurationFormat site
  - Intl.DurationFormat specification
- About V8
  - V8 Official Doc
    - Checkout V8 Source
    - Using Git to get V8
    - Building w/ GN
    - Testing in V8
  - V8 Embedder Guide
  - V8 Development How To (Additional Info)
  - Search Code in V8
- About ICU4C
  - ICU4C MeasureFormat Class Reference
- About ECMAScript
  - ECMA-262 ECMAScript2019 Specification
  - ECMA-402 ECMAScript 2017 Internationalization API Specification
  - Test suite for ECMA-262 https://github.com/tc39/test262
- About git
  - git Tutorial

Design Review

Review Notes

Appendix A Previous Design of JSDurationFormat

Duration Stage 2 Shape of API

The Following are Obsoleted Because the Changes of the Spec Text prior ot Stage 3
The information in this section is obsolete and only archived for historical purposes.

Design of JSDurationFormat

We research how to implement Intl.DurationFormat by prototyping with three different approaches to delegate the task to ICU. None of the three prototypes completely implement the June 1, 2020 specification text by using pre-existing functionalities in ICU 69.1. Each prototypes explored some shortcomings in ICU implementation and/or defects. However, we found Option C might be the easiest way to complete the implementation requiring the minimum changes to ICU 70.1:

- **Option A**: Use MeasureFormat::formatMeasures API
- **Option B**: Use LocalizedNumberFormat with the “x-and-y” unit support
- **Option C**: Use ListFormatter with an array of LocalizedNumberFormat combination

Design of Option A - MeasureFormat

This option was prototyped in 2762664 after I searched the ICU code and confirmed with Markus and Shane that MeasureFormat is currently the only ICU class support the “dotted” (aka “digital”) style. It is based on the formatMeasures() method in MeasureFormat. The function signature is

```
UnicodeString    formatMeasures (const Measure *measures, int32_t measureCount,
&                UnicodeString &appendTo, FieldPosition &pos, UErrorCode &status)
```

Notice, the method expects the caller to pass in an array of Measure, but Measure does not have a public default constructor, and therefore we need to use dummy Measure to initialize the array on the stack.
MeasureFormat support the following width: UMEASFMT_WIDTH_WIDE, UMEASFMT_WIDTH_SHORT, UMEASFMT_WIDTH_NARROW, and UMEASFMT_WIDTH_NUMERIC.
So we can map the style in the proposal to the width easily as below:

- “long” : UMEASFMT_WIDTH_WIDE
- “short” : UMEASFMT_WIDTH_SHORT
- “narrow”: UMEASFMT_WIDTH_NARROW
- “dotted” (aka “digital”): UMEASFMT_WIDTH_NUMERIC

For this approach, we need to store locale, style, numberingSystems and fields (or smallestUnit/largestUnit) into the JSDurationFormat object since MeasureFormat does not provide us a way to query such information. We also need to store the MeasureFormat pointer into the JSDurationFormat object.

Open Issues with this option/approach

1. FormatMeasure current does not have a method to return a FormattedValue, which is needed to implement formatToParts. Therefore, we need to add a new method to MeasureFormat to do so. See Appendix A for a draft of such a proposal. We will propose to ICU-TC if we decide we like to take Option A.
2. The current behavior UMEASFMT_WIDTH_NUMERIC handles the combination of “hours”, “minutes” and “seconds” and then falls back to another style. Depending on the change of the spec text, we may not be able to fully implement the “digital” style by using it and may need to change the ICU implementation to do so.

Design of Option B - LocalizedNumberFormatter w/ “X-and-Y” unit

This option was suggested by Shane. I decided to prototype it to verify the claimed functionality on the surface is enough to support the implementation of Intl.DurationFormat spec text. It was prototyped in 2775300 .

The basic idea of this approach is to construct a mixed unit with “X-and-Y” syntax and use LocalizedNumberFormatter to implement it. So if we like to format “hours”, “minutes”, “seconds”, we will construct the unit as “hour-and-minute-and-second” and then call MeasureUnit::forIdentifier() to construct the MeasureUnit and pass it to the LocalizedNumberFormatter in the constructor. We also map the style to UNumberUnitWidth as below. Notice the third column of UListFormatterWidth is not used in this approach but Option C only. We listed it together here since the mapping of UNumberUnitWidth is the same for Option B and C:

style	UNumberUnitWidth used by LocalizedNumberFormatter	UListFormatterWidth used by ListFormatter
“long”	UNUM_UNIT_WIDTH_FULL_NAME	ULISTFMT_WIDTH_WIDE
“short	UNUM_UNIT_WIDTH_SHORT	ULISTFMT_WIDTH_SHORT
“narrow”	UNUM_UNIT_WIDTH_NARROW	ULISTFMT_WIDTH_NARROW

For this approach, we need to store the locale and the pointer of the LocalizedNumberFormat into the JSDurationFormat object. We also store the fields (smallestUnit/largestUnit) to the JSDurationFormat object since we need this information during the format and parsing out such information from the skeleton could be too slow. We could parse the numberingSystems and style

information from the skeleton during the call of resolvedOptions() and therefore we do not need to store them into JSDurationFormat object.

In the implementation of the format() method, we need to convert the values of each unit to the largestUnit and aggregate them together since the LocalizedNumberFormatter has only methods to receive ONE numeric parameter (in either the form of int64_t, double, or **StringPiece** , but still only for ONE numeric value). Therefore, we need to convert the values between all 10 duration field units before calling the ICU API of LocalizedNumberFormatter

- Years
- Months
- Weeks
- Days
- Hours
- Minutes
- Seconds
- Milliseconds
- Microseconds
- Nanoseconds

In our prototype, we assume the following for conversion:

- 1 year = 365 days
- 1 month = 30 days
- 1 week = 7 days
- 1 day = 24 hours
- 1 day = 24 * 60 minutes
- 1 day = 24 * 60 * 60 seconds
- 1 day = 24 * 60 * 60 * 1000 milliseconds
- 1 day = 24 * 60 * 60 * 1000 * 1000 microseconds
- 1 day = 24 * 60 * 60 * 1000 * 1000 * 1000 nanoseconds

Open Issues and Difficulties by using this option/approach

We experienced four major difficulties in this approach, two from the defects of ICU implementation:

1. ICU-21544 unit format in number formatter returns U_INTERNAL_PROGRAM_ERROR with "year-and-" (except month) and "month-and-". This means in general we CANNOT use LocalizedNumberFormatter to support any combination of “years” and “months” EXCEPT the following three:
    - {largestUnit: “years”, smallestUnit: “years”}
    - {largestUnit: “years”, smallestUnit: “months”}
    - {largestUnit: “months”, smallestUnit: “months”}
2. ICU-21547 nextPosition() of FormattedNumber of units with "-and-" is buggy: LocalizedNumberFormatter does support the method to return FormattedNumber.

However, we discover the implementation is both buggy when the unit is mixed and the information it returns may not fit what the evolved proposal required.

3. We also believe since we are forced to convert all the values to the largest unit, we may produce rounding error and loss of precision with much higher chances. If we need to use int64 to represent years in nanoseconds, we could only express up to 292 years without overflowing the lowest 63 bits in int64 . Of course we may use StringPiece to express very big integers but then it carries the burden to switch between different representations depending on the value and largestUnit/smallesUnit we get and may have hard time to perform all the necessary calculations with some of the acceptable representations. We are not sure how to change that in LocalizedNumberFormat to improve since by definition, LocalizedNumberFormat should be used to format a number, but not two or more together.
4. LocalizedNumberFormatter does not support the “dotted” (aka “digital” style). It is not clear what changes are needed to support it in LocalizedNumberFormat yet.

Design of Option C - ListFormatter + array of LocalizedNumberFormatter

This option was prototyped in 2776518 after I discovered all the issues in Option B. It constructs an **DurationFormatInternal** object to handle each style. There are two subclasses of DurationFormatInternal:

- **ListDurationFormat**: a concrete implementation of **DurationFormatInternal** to handle “long”, “short” and “narrow” styles.
- **DigitalDurationFormat**: a concrete implementation of **DurationFormatInternal** to handle the “dotted” (aka “digital”) style. Due to the evolution of the specification text, It is not yet 100% clear how we would implement this class. See Option C-1 to C-4 below for possible approaches.

Design of ListDurationFormat

We uses an array of LocalizedNumberFormatter, each response to format one and only unit, and use the limited, but functional, unit support in LocalizedNumberFormatter to format each units, and then use one ListFormatter to combine the results of an array of FormattedNumber returned by the array of LocalizedNumberFormatter. We can easily implement the formatToParts by aggregating the information we get from the result of nextPosition() from the FormattedList returned by ListFormatter and from the array of FormattedNumber returned by LocalizedNumberFormatter.

Open Issues with this option/approach

1. How to implement **DigitalDurationFormat** to support the “dotted” (aka “digital”) style: To support format(), we could just call the MeasureFormat::formatMeasures(). However, we won’t be able to use it to implement formatToParts() for the same reason

of Option A. Depending on how the specification text evolves in this area, we may choose not to use MeasureFormat to implement it at all. Several possibilities:

a. **Option C-1**: Enhance MeasureFormat API to support FormattedValue as stated in Appendix A.
b. **Option C-2**: Enhance LocalizedNumberFormat to support the “X-and-Y” unit with new style.
c. **Option C-3**: Add a new formatter to ICU, which read the "durationUnits" to support that,
d. **Option C-4**: Implement inside v8 with public available methods of ICU 69.1, with the reading of "durationUnits" from ICU resources.

For this approach, we need to store the locale and the pointer of the **DigitalDurationFormat** into the JSDurationFormat object. The field information is stored in the **DigitalDurationFormat.** We also add method to DigitalDurationFormat to parse the skeleton of the LocalizedNumberFormat it hold onto to resolve the numberingSystems and style information during the call of resolvedOptions() and therefore we do not need to store them redundantly in the JSDurationFormat object.

Comparison of Option A, B and C

Below is a summary of the differences and issues between these prototypes, notice the reason we have difficulty to implement the “dotted” format is the spec text has not fully specify what the implement should do for the units other than “hours”, “minutes” and “seconds” if they are present in the input duration object.

	Option A	Option B	Option C
ICU Classes	MeasureFormat	LocaliedNumberFormat w/ “X-and-Y” unit support	ListFormatter and an array of LocalizedNumberFormat
CL of prototype	2762664	2775300	2776518
Lines of changes in v8	`+870 -3`	`+906 -5`	`+978 -7`
format()			
style: “long”, “short”, “narrow”	YES	PARTIAL [1]	YES
style: “dotted” (aka “digital”)	YES	NO	NO
formatToParts()			
style: “long”, “short”, “narrow”	NO	NO	YES
style: “dotted” (aka “digital”)	NO	NO	NO
ICU bugs discovered		ICU-21544 unit format in	

		number formatter return U_INTERNAL_PROGRAM_ ERROR with "year-and-" (except month) and "month-and-" ICU-21547 nextPosition() of FormattedNumber of unit with "-and-" is buggy.	
ICU feature enhancement required	ICU-21543 Add methods to return FormattedValue to MeasureFormat		
Unclear Open Issue		How to support “dotted” (aka “digital”)	How to support “dotted” (aka “digital”)

- [1] year and month can only combine with each other but not other units

Possible API proposal to ICU-TC for ICU 70.1 if we decide to take the Option A approach:

-

```
Add the following to public section of class U_I18N_API MeasureFormat : public Format { }

#ifndef U_FORCE_HIDE_DRAFT_API

   /**
    * Formats measure objects to a FormattedList, which exposes field
    * position information. The FormattedList contains more information than
    * a FieldPosition. An example of string in such a
    * FormattedList is 3 meters, 3.5 centimeters. Measure objects appear
    * in the FormattedList in the same order they appear in the "measures"
    * array. The NumberFormat of this object is used only to format the amount
    * of the very last measure. The other amounts are formatted with zero
    * decimal places while rounding toward zero.
    * @param measures      array of measure objects.
    * @param measureCount  the number of measure objects.
    * @param status        the error.
    * @return              A FormattedList containing field information.
    * @draft ICU 70
    */
  FormattedList formatMeasuresToValue(
           const Measure *measures,
          int32_t measureCount,
          UErrorCode &status) const;

#endif  // U_FORCE_HIDE_DRAFT_API
```