Strawman proposal for v0.5 EcmaScript i18n Surface

Acknowledgements

Some of this text is taken from previous documents and discussion, including the last meeting and the Google strawman.

Why v0.5?

For the EcmaScript i18n surface, there is some “low hanging fruit” that is reasonably straightforward to understand and adopt. Currently there’s a huge gap with the lack of an EcmaScript globalization experience. A v0.5 effort lets us get at the straightforward behavior without causing undue delay. Additionally it provides a backbone so that structure of EcmaScript i18n is stable, providing clarity to the future direction. By separating the “obvious” functionality from details that either require more investigation or are more controversial, we can provide basic support to the community without delay.

The proposed goal is to have a stable specification for a v0.5 standard ready for GA review by June, 2011. A goal is that implementations should be able to take a dependency on this v0.5 specification at that time.

Approach to the Surface

This document takes the consensus from the November meeting and removes those items that required further investigation and those that could be more controversial. The expectation is that those will be easily added to the v1.0 full extension.

There are some parts that must be completely understood however, even if the initial surface is small. In particular the entire architecture depends on the locale or context constructs used to create the rest of the objects. That basic infrastructure needs to provide support for, or at least be extensible to, the full i18n standard.

Concepts that can be added later which are ambiguous, difficult to support on some system(s) or controversial are, therefore, omitted for v0.5.

Goals

- Applications should be able to get a “user default” globalization experience.
- Applications need to be able to specify specific locales, perhaps with fallback.
- For both of these the application needs to be able to determine what the locale actually resolved to. (Eg, user default is French, asked for Hawaiian allowing fallback, got English). This is very important and fundamental to Microsoft.
- Allow selection of language, with regional variant, and an explicit region. (eg: Canadian English in the US).

- Surface should seem at least somewhat familiar to ICU users, maybe a superset/extension, but not completely radical.
- Needs to be vendor extensible.
- V0.5 must support the varied infrastructures that implementations may be built on top of. Eg: ICU, Windows NLS, .Net, Flash, etc.

Out of Scope

The following are out of scope to keep v0.5 on track and small. That doesn't preclude parallel discussions of the v1.0 behavior so long as that doesn't delay the v0.5 effort:

- Timezones introduce timezone ids, which are not currently standardized, and are inconsistently adopted. This v0.5 effort will not be able to support timezones.
- Resource loading / message formatting has significant variations by platform, which will likely make require more discussion and investigation, so it is omitted for v0.5. (Insofar as globalization data is a resource, implementations may use appropriate platform specific methods to retrieve data for v0.5).
- CLDR/Data Consistency – Platforms have different cultural data, so it is expected that implementations will differ in the returned i18n data. Though it may vary, data is reasonably expected to be appropriate for its application.
- Message Formatting lacks OS support in some platforms, making adoption difficult, so it will be omitted in v0.5.
- Parsing – Parsing isn't available on all platforms and is omitted in v0.5.
- UCA - the Unicode Collation Algorithm is not consistently adopted for all platforms, so standardization on the UCA behavior is out of scope for now. Collation behavior is expected to vary by platform and version.
- AvailableLocales/Languages/Locations – The "available" list could be very large, and it is likely that this concept would cause some debate, so, for now, the AvailableXXXX list functions are out of scope for v0.5.
- Unicode operations – Normalization, Casing, Unicode properties, etc. Most of these are straightforward, but a lack of discussion removes them from the v0.5 scope.
- Regular Expressions – Complex and not well discussed yet.
- Things that are reasonably straightforward to work around, to reduce the risk of ratholing causing delays to v0.5.
- Language and Region objects – There are differing perspectives around these, and we can avoid using objects by passing strings to the locale constructor.

Design

This proposal takes the form of the original Google strawman, considers the discussion of the last meeting, and removes any complicated features, controversial features, or things that can be avoided for now.

V0.5 keeps the general idea of having individual objects for each piece of functionality. Options are provided in the constructors to specify behaviors. The structure is regularized so that each constructor takes a single options parameter, which contains each of the parameters. Using an options basket also allows for simple vendor specific extensions if necessary.

Proposed API

Locale

The locale class identifies a set of conventions that are used in other i18n classes. In this API it's an object so that validation and canonicalization does not have to be repeated.

Locale Constructor

aLocale = new Locale(options);
Creates a Locale object from the given options. If there is no match or no options provided it returns the default locale. The default locale should be appropriate for the user, however the choice of default is up to the implementation).

Parameters:

{Object} options – Options to select/construct the locale. If options are omitted we use a default locale.

Errors:

May throw an error if conflicting options are passed.

Options Property

options = aLocale.options;
aLocale.options = { *option*:*value* };
The options that describe the locale. Changing options could cause additional canonicalization checks or change the behavior of the locale.

The options allow creation of a locale from a BCP47 style locale name, or an explicit list of languages and a region. If information is missing, then the default locale is used.

Available Options:

{String} localeName
A BCP47/CLDR style locale name, possibly including an implicit region.

{Array} languageNames
List of languages to get a compatible locale from.

{String} regionName
Explicit region for this locale. Note that this region selection indicates regional preferences, not linguistic variants.

localeName property

myString = aLocale.localeName;
The localeName property returns the actual locale name resolved to from the inputs, which may be the default locale.

Explicitly out of scope for Locale for v0.5:

availableLocales – could be large
maximizedLocale – not all frameworks have the info.

minimizedLocale – not all frameworks have the info.
displayName – most apps don't need it and could have some churn on the signature/behavior.
language & region accessors – There is discussion about language & region objects usefulness, so those are omitted for now. Instead the options constructor accepts string names.

Locale Examples:

```
var locale = new Locale({ localeName: "de-DE-u-co-phonebk");
var locale2 = new Locale({ languageNames: ["haw", "en-CA", "en"],
                           region: "US" };
var locale2Name = locale2.localeName;   // could return "haw-US", "en-US", or a default.
```

Collator

The collator object encapsulates collation logic.

Collator Constructor

aCollator = new Collator(options);
Creates a Collator object from the given options. If there is no match or no options provided it returns the default collation for the default locale. The default collation should be appropriate for the user, however the choice of default is up to the implementation).

Note that the returned collation behavior will be dependent on the host framework or versions, and will likely differ between machines. As such, only general behavior should be expected from the v0.5 collator.

Parameters:

{Object} options – Options to select/construct the collator. If options are omitted we use a default collator.

Errors:

May throw an error if conflicting options are passed.

Options Property

options = aCollator.options;
aCollator.options = { *option*:*value* };
The options that modify the collator. If insufficient options are specified, a default collator is used.

Available Options:

{locale Object} locale
An optional locale object to use for this collator. If not specified the default locale is used.

{int} strength
An ICU/UCA style strength for the collation. It can have values 1 to 5.
Note/Warning: Framework support for strength will differ greatly, so some variation in behavior on different machines should be expected.

compare method:

{int} myCollator.compare(s1, s2);
This compares two strings.

Return Value:

< 0 if s1 < s2
== 0 if s1 == s2
> 0 if s1 > s2

Collator Examples:

```
var locale = new Locale({ localeName: "de-DE-u-co-phonebk");
var col = new Collator({ locale: locale });
someStringArray.sort(col.compare);

var col2 = new Collator({ locale: locale, strength: 5 });
someStringArray.sort(col2.compare);
```

NumberFormat

The NumberFormat object provides number formatting.

NumberFormat Constructor

var nf = new NumberFormat(options);
Creates a NumberFormat object from the given options. If there is no match or no options provided it returns the default number format for the default locale.

Parameters:

{Object} options – Options to select/construct the NumberFormat. If options are omitted we use a default NumberFormat.

Errors:

May throw an error if conflicting options are passed.

Options Property

var options = myNumberFormat.options;
myNumberFormat.options = { *option*:*value* };
The options that modify the number format. If insufficient options are specified, a default number formatter is used.

Available Options:

{locale Object} locale
An optional locale object to use. The number format will be appropriate for this locale.

{string} style
The style of number formatting to use: “decimal”, “currency”, or “percent”

- decimal – 1234.5
- currency – $1234.50
- percent – 76%

format method

{string} myNumberFormat.format(*value*)
Formats a value according to the specified NumberFormat behavior.

Return Value:

A string with the formatted value.

Explicitly out of scope for v0.5 NumberFormat

- Parsing
- Patterns
- Scientific style
- ISO currency style
- Currency object
- Currency symbol

NumberFormat Examples

```
var nf = new NumberFormat( { style:"decimal" } );
nf.format(12.30);   // returns 12.3
var de = new locale({localeName: "de"});
nf.options = { locale:de, style:"currency"};
nf.format(12.3);  // returns €12,30
```

DateTimeFormat

The DateTimeFormat object provides number formatting.

DateTimeFormat Constructor

var dtf = new DateTimeFormat(options);
Creates a DateTimeFormat object from the given options. If there is no match or no options provided it returns the default number format for the default locale.

Parameters:

{Object} options – Options to select/construct the DateTimeFormat. If options are omitted we use a default DateTimeFormat.

Errors:

May throw an error if conflicting options are passed.

Options Property

var options = dtf.options;
dtf.options = { *option*:*value* };
The options that modify the date time format. If insufficient options are specified, a default date time formatter is used.

Options specify the default locale to use, and the style of date or time. If both date and time styles are passed, then the output of format will contain both date and time.

Available Options:

{locale Object} locale
An optional locale object to use. The date time format will be appropriate for this locale.

{string} dateStyle
The style of date formatting to use: "short", or "long". Note that in v0.5 these may differ slightly from the CLDR meanings depending on the implementation.

- short – 1/18/2011 – short date form, era only if necessary for the calendar (eg: Japanese).
- long – Tuesday, January 18, 2011 – long date form, era only if necessary for the calendar

{string} dateSkeleton
A "skeleton" describing fields desired in the output date style. For v0.5 only a few skeletons are supported:

- yMd – Same as short dateStyle.
- yyyyMMMMdEEEE – Same as long dateStyle, include day of week.
- yyyyMMMMd – Long date without day of week, era only if necessary.
- yMMM – Year/month pattern.
- MMMd – Month/day pattern.

{string} timeStyle
The style of time formatting to use: "short", or "medium". Note that in v0.5 these may differ from their CLDR meanings.

- short – 10:00 AM
- medium – 10:00:02 AM

{string} calendarName
Optional calendar to use, otherwise use the calendar of the locale. Supported names:

- gregorian
- islamic – alias hijri? – Astronomical Islamic (Arabic) Calendar
- islamic-civil – alias um-al-qara? – Civil Islamic (Arabic) Calendar
- hebrew – Traditional Hebrew calendar
- japanese – Japanese Imperial Calendar, similar to Gregorian, but with emperor eras.
- chinese – traditional Chinese calendar
- buddhist – alias thai-buddhist or thai
- roc – alias taiwan – ROC calendar
- korean – Korean era calendar

prototype.format method

{string} myDateFormat.format(*value*)
Formats a value according to the specified DateTimeFormat behavior.

Return Value:

A string with the formatted value.

static format method

{string} DateTimeFormat.format(*options*, *value*)
Simplified static method for easily getting a default formatted value.

> Note: If the working group reaches consensus regarding this static method, similar static methods for other classes, e.g., NumberFormat, could also be considered.

options

Optional options to use when formatting the value. Same values as the instance version.

value

The value to format.

Return Value:

A string with the formatted value.

Explicitly out of scope for v0.5 DateTimeFormat

- Parsing
- Patterns
- Timezone – WARNING: A lack of timezone means that applications need to be very careful about their UTC vs local time conversions and display.
- Additional Skeletons
- Full Date Style
- Medium Date Style
- Full Time Style
- Long Time Style
- Coptic Calendar
- Ethiopic Calendar
- Persian Calendar

DateTimeFormat Examples

```
var df = new DateTimeFormat( { dateStyle:"short" } );
df.format(new Date());   // returns 1/18/2011 assuming us locale
var de = new locale({localeName: "de"});
df.options = { locale:de };
df.format(new Date());  // returns 18/1/2011
```

var df2 = new DateTimeFormat({ dateStyle:"long", timeStyle:"short" });
df2.format(new Date()); // returns “10:00 AM 18 January, 2011”

DateTimeFormatSymbols

DateTimeFormatSymbols Constructor

var dtf = new DateTimeFormatSymbols (options);
Creates a DateTimeFormatSymbols object from the given options. If there is no match or no options provided it returns the default number format for the default locale.

Parameters:

{Object} options – Options to select/construct the DateTimeFormatSymbols. If options are omitted we use a default DateTimeFormatSymbols.

Errors:

May throw an error if conflicting options are passed.

Options Property

var options = dtf.options;
dtf.options = { *option*:*value* };
The options that modify the date time format symbols. If insufficient options are specified, default values are used.

Options specify the default locale to use.

Available Options:

{locale Object} locale
An optional locale object to use. The date time format will be appropriate for this locale.

{string} calendarName
Optional calendar to use, otherwise use the calendar of the locale. Supported names:

- gregorian
- islamic – alias hijri? – Astronomical Islamic (Arabic) Calendar
- islamic-civil – alias um-al-qara? – Civil Islamic (Arabic) Calendar
- hebrew – Traditional Hebrew calendar
- japanese – Japanese Imperial Calendar, similar to Gregorian, but with emperor eras.
- chinese – traditional Chinese calendar
- buddhist – alias thai-buddhist or thai
- roc – alias taiwan – ROC calendar
- korean – Korean era calendar

getMonths method

{string[]} mySymbols.getMonths(options)
Returns an array containing the requested month names. These are in the language of the input locale or a default language. If the strings aren’t available in the input locale, default or other fallback values may be returned.

options
Modifiers to the month names being returned. Standalone wide strings are the default values if no options are passed.

{string} width – "wide" or "narrow". Eg: "January" or "Jan"

{string} context – "nominal" or "genative"..

Return Value:
An array of strings with the month names. The array length may be 12 or 13 months depending on the calendar.

getDaysOfWeek method
{string[]} mySymbols.getDaysOfWeek(options)
Returns an array containing the requested days of the week. these are in the language of the input locale or a default language. If the strings aren't available in the input locale, default or other fallback values may be returned.

options
Modifiers to the day names being returned. Standalone wide strings are the default values if no options are passed.

{string} width – "wide", "narrow", or "abbreviated". Eg: "Monday", "Mon", or "Mo"

Return Value:
An array of strings with the day names.

Explicitly out of scope for v0.5 DateTimeSymbols
- Setting of symbol values
- Any other values from ICU
- Abbreviated month name widths. (Not supported on all platforms)
- Variations of names depending on actual date.
- Returning "current" values for "now".
- Genitive forms of days of week.

DateTimeSymbol Examples
```
var de = new locale({localeName: "de"});
var dts = new DateTimeSymbol({ locale:de });
var months = dts.getMonths({width:"narrow"});
var days = dts.getWeekdays({width:"abbreviated"});
```

Explicitly Out of Scope:
For v0.5 the following are also explicitly out of scope:
- Calendar object – uses names instead of calendar objects right now to reduce complexity
- First day of week – Since there's no calendar object.
- Currency objects

- NumberFormatSymbol objects.

Extensibility:

ECMAscript is very extensible by its nature. Vendors may implement their own private behaviors or properties. This extensibility can be used to support non-standard vendor-specific functionality, or to explore functionality that was left out of v0.5.

Options

All of the constructors accept options to modify their behavior. Vendors may use a vendor-specific prefix to add their own options. Implementations should ignore unrecognized options.

Functions

Any vendor specific or experimental methods must be prefixed with a vendor prefix. It should be assumed that these will cause failures if used on a different vendor's platform.