iLib 1.0 Javascript Tutorial

This document explains what you need to know in order to understand and use iLib on your web site or in any other Javascript program.

1.0 What is iLib?

iLib is a library of internationalization (i18n) routines for Javascript. It is modeled after similar routines in the International Classes for Unicode (ICU) package for C, C++, and Java. It provides a way to internationalize your code without needing to write these classes from scratch.

2.0 Why iLib?

Javascript as implemented in browsers and standalone engines like Nodejs is purposely small to make sure it doesn't fill memory with libraries of code that are seldom used. Much of the time, you can get away with returning already-formatted locale-sensitive data from the web server to the client, so internationalization routines are often not even needed in the browser.

Unfortunately, despite these work-arounds, there are times when you do need internationalization in your Javascript code anyways. And yet general i18n libraries don't exist -- until iLib of course!

For example, let's say you use an external service to get something done on your web site. Not every company has done internationalization for their API and you could get data back from the call that is not formatted for the user's locale, or perhaps the data is in a normalized canonical form. Now if you make an AJAX call to that web service, you will need to format the return data in your Javascript, as your own web server is not involved and cannot format it for you. This is a job for iLib!

Let's say the other company gives you back a JSON object with a "createdate" property with the value being an integer that encodes a unix time for the creation date of the object. ("Unix time" is the number of seconds since midnight Jan 1, 1970.) In this case, the regular Javascript libraries do include a way of formatting dates. To do that, you can convert the integer to a Javascript Date object and then format it for display in the UI. The code would look something like this:

```
var d = new Date(webserviceobject.createdate);
var n = d.toLocaleString();
```

The resulting output might be something like this:

```
Mon Apr 02 2012 16:35:26 GMT-0700 (PDT)
```

At first blush, that seems sufficient. But when you look more closely, there are a number of problems with that. What if you want to use a different locale than the browser is using? The code above can only format dates in that one locale. What if you want a different format for the date string, such as a long version or a short version? The above code only does the full version. What if you want different date components such as only the month, day, and year without the day-of-week? The above code always does everything. What if you want to display the date in a non-Gregorian calendar? The above code is Gregorian-only. What if you want to

format the date in a locale that the browser doesn't know anything about? The browser only has one locale at a time.

With iLib, all of those problems are solved. Let's say the user's browser is running in English, but they signed up to your web site and picked German/Germany as their locale. Let's say you are displaying a date/time in a column in a table, and to save screen real estate, you would like to show the short version of the date and time, and don't want the "extra" date components such as the day-of-week or timezone. The equivalent iLib code would look like this:

```
var d = new ilib.Date.GregDate({
     unixtime: webserviceobject.createdate
});
var fmt = new ilib.DateFmt({
     locale: "de-DE",
     type: "datetime",
     length: "short",
     date: "dmy",
     timezone: "Europe/Berlin"
});
var n = fmt.format(d);
```

The output would be:

```
02.04.12 16:35
```

Aha, that's the output we were looking for!

One big advantage of the code above is that the formatter object is separate from the date object. That means you can instantiate a formatter and pass it around in your code. The rest of the code does not need to know about locales or output styles, etc. It just knows it needs the formatter object to format the date objects correctly.

Other parts of iLib are similarly flexible. Every class can take a locale specifier or a locale instance as part of its parameters, and is more configurable than any standard Javascript classes that they may replace.

iLib was designed to be self-contained. That is, it does not rely very much on the standard Javascript libraries or the capabilities of the engine upon which it is running. Consequently, iLib can be used in nodejs, v8, or on any web browser with no functionality differences.

3.0 What Can it Do?

iLib has a number of things it can do right now, and more functionality is coming in future releases. Currently, it has classes that can:

- Format date objects as a date, time, or date and time. Parsing is not supported yet.
- Format two date objects as a date/time range.
- Format a length of time as a duration. This is different than a range in that durations can have parts that are larger than the limits for normal dates and do not have a specific start and end date/time. For example, a duration may be “40 days”, but a date range for the same amount of time might be “Feb 22 - Apr 2, 2012”. Forty days is too large for a month, but as a duration, it can make sense.
- Give information about a calendar type. Currently Gregorian, Julian, and Islamic Civil calendars are supported, with more to come.
- Give information about a time zone, including whether daylight savings time is observed in the locale and if so, when the start and stop date/times are.
- Format and parse numbers
- Format and parse currency amounts
- Get information about currencies
- Format and parse percentages
- Get translated versions of your Javascript strings.
- Get information about character types for any Unicode character. This is similar to the C library CType functions.

iLib also includes a tool to assemble only those parts of the code you want to use on your web site into a single, compressed file. If you don’t use the CType routines in your Javascript, you don’t have to include them or their data in your assembly of the Javascript files, which makes the overall JS file smaller and quicker to transfer across the Internet. The assembly tool also has a way to only include the locale data for the locales that your web site supports, which again makes the overall JS file smaller. The assembly tool is even general enough that you can use it with your own Javascript files.

4.0 Using iLib

In this tutorial, we will slowly internationalize a very simple web page. This web page displays a number of pieces of locale-sensitive data. The initial and final HTML files, start1.html and end1.html, can be found along with this tutorial in the iLib package. You can open these files with any browser and test them out for yourself.

4.1 Date Formatting

To start, we have the following simple HTML file (start1.html):

```
<html>
<script>
var now = new Date();  // current date
var duration = 36.5*60*60*1000; // 36 hours and 30 minutes in
milliseconds
var soon = new Date(now.getTime()+duration); // 36.5 hours from
now

function formatStuff() {
    var datetime = document.getElementById("datetime");
    datetime.innerHTML = now.toLocaleString();

    var range = document.getElementById("range");
    range.innerHTML = now.toLocaleString() + " - " +
soon.toLocaleString();

    var durationElement = document.getElementById("duration");
    var hours = Math.floor(duration / (60 * 60 * 1000));
    var minutes = (duration / (60 * 60 * 1000) - hours) * 60;
    durationElement.innerHTML = hours + ":" + minutes;
}
</script>
<body onload="javascript:formatStuff();">
The date is <div id="datetime"></div>
The date range is <div id="range"></div>
The duration is <div id="duration"></div>

</body>
</html>
```

Note that the “stuff” that is formatted into the div elements in the body is not internationalized properly. If the user’s browser is running in English, the output would be:

```
The date is:
Mon Apr 02 2012 18:09:37 GMT-0700 (PDT)
The date range is
Mon Apr 02 2012 18:09:37 GMT-0700 (PDT) - Wed Apr 04 2012 06:39:37 GMT-0700 (PDT)
The duration is
36:30
```

That output is sort of ugly! There are too many parts to the date strings. If we switched the locale to Italian, for example, it would just be plain wrong because the proper Italian format uses a period between the time elements instead of a colon, and the day comes before the month, and the time zone is CET. There are lots of problems!

Now, let's introduce iLib formatter objects to format these things instead:

```
<html>
<script src="../../js/src/iliball.js"></script>
<script>
var now = new ilib.Date.GregDate();  // current date
var duration = 36.5*60*60*1000; // 36 hours and 30 minutes in
milliseconds
var soon = new ilib.Date.GregDate();
soon.setTime(now.getTime()+duration); // 36.5 hours from now

function formatStuff() {
    // create default formatters for now
    var fmt = new ilib.DateFmt();
    var rangeFmt = new ilib.DateRngFmt();
    var durFmt = new ilib.DurFmt();

    var datetime = document.getElementById("datetime");
    datetime.innerHTML = fmt.format(now);

    var range = document.getElementById("range");
    range.innerHTML = rangeFmt.format(now, soon);

    var durationElement = document.getElementById("duration");
    var hours = Math.floor(duration / (60 * 60 * 1000));
    var minutes = Math.floor((duration / (60 * 60 * 1000) -
hours) * 60);
    durationElement.innerHTML = durFmt.format({
        hours: hours,
        minutes: minutes
```

```
    });
}
[...]
```

The first interesting thing to note is that there is a new script tag that includes the iLib library. The download package comes with a kitchen-sink version of the iLib JS files ("iliball.js") where all code and all data for all locales are assembled together in the right order in one big file. You can rebuild this iliball.js file yourself. The chapter below on "Assembling a Javascript File" gives details on how it works and how you can generate smaller copies of the iLib files with only the stuff you need in it, and even how to generate the much smaller compressed version of it. You can even use the assembly tool with your own javascript files.

The second thing to note is that the Javascript *Date* instances have been replaced with *ilib.Date.GregDate* instances. The difference between them is that the *GregDate* class has a number of methods that *Date* does not have which help you convert between calendars, figure out which day of the week a date or relative date is, or what the day of the year or the week of the year is for the date.

More on date objects later. Let's move on to the formatters. Unlike the Javascript *Date* class, formatters in iLib are separate classes from the date objects. This follows the Java/ICU paradigm. Separating them has a number of advantages:

- Formatters can be instantiated once and reused over and over again. The work of figuring out what the actual format template is for the options only occurs once. For example, if you had a table of database results, and each result had a date column, you could format each of those dates with the same formatter.
- Code that formats strings does not have to know about locales or a whole host of other parameters that go into instantiating a formatter class, and therefore this data does not need to be passed in to that code. You just pass the formatter itself.
- Formatters can handle different calendars. With the Javascript *Date* class, the built-in formatter is hard-wired to the Gregorian calendar.
- Multiple formatters can be instantiated with different options so that different forms of date/time formats can be formatted at the same time.

Finally, note that for all the formatters have a format method which take various combinations of date objects as parameters. That is where all the heavy lifting is done to generate the output strings.

The output from the above HTML file would look like this:

```
The date is:
```

```
4/3/12
The date range is
4/03/12 10:03pm - 4/05/12 10:33am
The duration is
36:30
```

The default for each of the formatters is the “short” length for the locale. That explains why this output is much more succinct than the output from the generic Javascript classes.

4.2 Playing with the Date Formatter Parameters

Now, let’s say that you wanted to modify the date formats because you don’t like the “short” output above. Let’s say you wanted to see the date formats in Italian, you wanted the long format instead of the short, and you want to see the time as well. Also, let’s say you also want to see the day of the week too.

Here is how you would modify the above code. (Note that here and in the rest of this tutorial, we only give the parts of the example that have changed from the previous example above.)

```
[...]
    var fmt = new ilib.DateFmt({
        locale: "it-IT",
        length: "long",
        type: "datetime",
        date: "wdmy"
    });

    var rangeFmt = new ilib.DateRngFmt({
        locale: "it-IT",
        length: "long"
    });

    var durFmt = new ilib.DurFmt({
        locale: "it-IT",
        length: "long"
    });
[...]
```

The first thing to note is that the formatter is controlled by options given to the constructor. The formatter is immutable after it is created, so it always formats the same way after that.

There is a locale parameter to each constructor. This overrides the locale of the browser or JS

engine. A significant number of people around the world use US English browsers and software because it is seen as the latest version, even if there are localized versions of the software available. Despite that, users still want to see their web sites in their own language, setting up a cross-language situation. To get the built-in Javascript *Date* class to do the right thing means that you either have to set the locale for the whole JS engine instance, or you just pass in the locale to each method. The latter method allows for more flexibility, but unfortunately, the generic Javascript classes do not support that.

The locale parameter to iLib formatters can be specified as a locale specifier or as an *ilib.Locale* instance. A locale specifier is simply a string that identifies the locale. The format is simple: 2 lower case characters that identify the language (see the ISO 639 standard) followed by a dash followed by two upper case letters that identify the country (see the ISO 3166 standard). There is more to it than that, but for now that is all you need to know. An *ilib.Locale* instance is not necessary here either, as you are not doing anything to manipulate the locale itself.

The locale determines a number of things, such as the format characters, the order of components, and the use of the 12- or 24-hour clock. Most of those can be overridden by other constructor parameters though.

The second thing to note is the length parameter, which is common to all three formatters. This specifies the length of the desired format. Each locale specifies a short, medium, long, and full format. In the short format, numbers are often used for month names. In the long format abbreviations are used for months. In the full format, entire month names are used. Other differences occur as well based on the length.

The parameters to the constructor of the date formatter object also include the “type” and “date” properties.

The “type” property specifies that this formatter should format both the date and the time together in the right order for the locale. Other valid values are “date” or “time” alone if those are the only parts you want.

The “date” parameter specifies which components of the date format to use. Each letter specifies one component such as the month or year. In this case, we want all of the components, so “w” for day of Week, “d” for Day, “m” for Month, and “y” for Year. You can give any combination of letters in any order to specify the components. The order that the components appear in the output is determined by the conventions of the locale, not by the order of letters in this parameter.

Please note that some combinations of components do not make much sense and so they are not supported in the format data. For example, if the date components are set to “wy”, then you might expect output like “Sunday 2012”, which is not that useful unless perhaps you are writing the titling to a comedic movie. The data for pretty much every locale does not support a format

that only contains the “wy” components, even though it theoretically could. However, “wdm” is supported, so you could format a date that looks like “Wednesday March 4” where the year is left out. You can use this, for example, to format someone’s yearly birthday in the right order for the locale using the components “md”. Example: this might be “August 23” in the US and “23 August” in Canada and the UK.

Finally, the output from the above code is:

```
The date is:
Mar 03 apr 2012 22.23
The date range is
03 22.23 - 05 10.53 apr 2012
The duration is
36.30
```

The locale has specified that the date comes before the time, date numbers smaller than 10 have leading zeros, day of week names have a capital letter whereas month names are written in lower-case letters, the date number comes before the month and then the year, the time separator char is a period, and the time is formatted with the 24-hour clock.

There are lots more things you can tweak in the formatter constructor options. See the API reference documentation for more details on all the specifics.

4.3 Translation

At the core of any globalization effort is the translation of text. Many people erroneously think that translation is all there is to globalization, and they don’t know about the internationalization of code or the localization of other, non-text resources. The fact that you are reading this documentation means that you already know better than that! This section, however, will focus on the translation support in iLib.

Translating Javascript strings is often done with some ad-hoc methods such as creating homebrew classes to load strings, or even creating multiple copies of the Javascript source files, one for each language. Without tools, the strings have to be extracted by hand. iLib allows your English-only engineers write their code as normal, inserting English strings in their Javascript where necessary. If you have a commercial license, you can use the iLib localization tool with your JS sources. If you don’t, you can externalize strings using tools such as the Eclipse IDE. Strings can be extracted from the JS or HTML source files, translated, and re-inserted into an assembled Javascript file that is included in a script tag in the HTML. Then, using the provided *ilib.ResBundle* class, you can load the translated strings and display them properly.

Looking at our example from the previous chapter, we can see that the strings in the HTML are not translated. Although the localization tool can also help you translate HTML files, this chapter

is going to focus on translating Javascript strings. Let's move the strings from the HTML into the Javascript portion of the file:

```
[...]
    var datetime = document.getElementById("datetime");
    datetime.innerHTML = "The date is " + fmt.format(now);

    var range = document.getElementById("range");
    range.innerHTML = "The date range is " +
rangeFmt.format(now, soon);


    var durationElement = document.getElementById("duration");
    var hours = Math.floor(duration / (60 * 60 * 1000));
    var minutes = Math.floor((duration / (60 * 60 * 1000) -
hours) * 60);
    durationElement.innerHTML = "The duration is " +
durFmt.format({
        hours: hours,
        minutes: minutes
    });
}
[...]
```

This will give similar output as before, but because the strings are now inside the div tag, the string and the formatted dates will all appear on the same line.

To modify the above code for translation with iLib, we would make it look like this:

```
[...]
soon.setTime(now.getTime()+duration); // 36.5 hours from now

var res = new ilib.ResBundle({
    locale: "it-IT",
    name: "ilibtutorial",
    type: "html"
});

function formatStuff() {

[...]
    var datetime = document.getElementById("datetime");
    datetime.innerHTML = res.getString("The date is ") +
```

```
fmt.format(now);

    var range = document.getElementById("range");
    range.innerHTML = res.getString("The date range is ") +
rangeFmt.format(now, soon);

    var durationElement = document.getElementById("duration");
    var hours = Math.floor(duration / (60 * 60 * 1000));
    var minutes = Math.floor((duration / (60 * 60 * 1000) -
hours) * 60);
    durationElement.innerHTML = res.getString("The duration is
") + durFmt.format({
        hours: hours,
        minutes: minutes
    });
}
[...]
```

Here we see the introduction of the new class *ilib.ResBundle*. The *ResBundle* class is the one that finds and loads string resources. Note that the strings are still concatenated with a plus operator, which is bad, but we'll deal with that in the next section below.

The main workhorse of the *ResBundle* class is the *getString()* method. This searches for a translation of the given source string and returns it. Typically, the source string is in English, but that does not have to be the case. You can use any language you like as the source language.

If the translation for the string does not exist, *getString()* will return the source string itself. This means that while you are developing your program or web site and the translations haven't been done yet, you will always get a reasonable string of some sort out of this method. Your German QA people may see the older parts of the UI translated to German and the new strings will be in English until the translations are ready and re-integrated into the project.

The *ResBundle* class will look for a Javascript object containing translations by looking for a variable with the name given in the name property augmented with the locale name. This way, you can have multiple sets of translations to different languages for the same bundle name. You can also have multiple bundles with different names so that they do not conflict with each other. This is useful if you have multiple JS source files included in your various HTML files, and you don't want the string resources to conflict.

When the strings are being loaded, the *ResBundle* class will also look up the "locale hierarchy" tree for other translations and merge them together in much the same way that Java does. For example, let's say you had your locale set to "fr-CH-govt" for the French-speaking part of Switzerland. The variant is for the Swiss government, as perhaps you have special translations

just for them. In this case, when you load in the resources for that locale, the resource bundle will first load any generic strings that are shared between all versions of French. ie. the “fr” locale. Then, it will load in all the translations for “fr-CH” for Swiss-French-specific translations. Any translation in “fr-CH” overrides the corresponding translations in “fr”, and may add a few of its own. Finally, “fr-CH-govt” is loaded in and any translations it provides overrides any existing translations.

Based on this merging, it is recommended that you put most translations into the set for the base language, and only override them with locale-specific translations where necessary. This way, any locale that you didn’t think of supporting before, but which shares a language with a locale you are supporting can also get mostly the right strings.

Example of a string that is overridden from a generic translation by a specific locale:

```
English source string: "Enter email address: "

French (fr) translation: "Entrez l'adresse e-mail : "

French Canadian (fr-CA) translation: "Entrez l'adresse de
courriel : "
```

In the absence of any other information when the localization tool extracts strings from a source file, it will assume that the name of the bundle will be the base name of the source file, unless it finds a *ResBundle* creation statement such as in the example above. In that case, it will assume all strings in this source file belong to that resource bundle. In our example, this is “ilibtutorial”.

The localization tool can also find special comments in the source file that name the resource bundle for the file. Example:

```
/* !resbundle mybundle */
```

This sort of comment works for any source file type when it is commented according to the normal commenting syntax of the programming language.

4.4 String Formatting

In the above example, the translated strings are concatenated together with dates/durations with a simple plus operator. This is a big no-no for translation! The reason is that in other languages, the date/duration part sometimes needs to be at the beginning or the middle of the string in order to be grammatically correct.

The solution to this problem is to do proper string formatting using the *ilib.String* class. The

ilib.String class is an extension of the built-in Javascript string class with a few extra methods to do the types of formatting that are needed for internationalization. The *getString()* method of a *ResBundle* already returns an instance of this class, so you don't have to wrap its output in an *ilib.String* constructor.

Here is how the above code should be modified to do formatting properly:

```
[...]
    var str;
    var datetime = document.getElementById("datetime");
    str = res.getString("The date is {date}");
    datetime.innerHTML = str.format({
        date: fmt.format(now)
    });

    var range = document.getElementById("range");
    str = res.getString("The date range is {range}");
    range.innerHTML = str.format({
        range: rangeFmt.format(now, soon)
    });

    var durationElement = document.getElementById("duration");
    var hours = Math.floor(duration / (60 * 60 * 1000));
    var minutes = Math.floor((duration / (60 * 60 * 1000) -
hours) * 60);
    str = res.getString("The duration is {duration}");
    durationElement.innerHTML = str.format({
        duration: durFmt.format({
            hours: hours,
            minutes: minutes
        })
    });
}
[...]
```

Note that the strings now contain replacement parameters enclosed in braces, such as "{date}" and "{range}". These are replaced by the values of the parameters to the *format()* method, and the resulting string is returned. Translators know not to translate replacement parameters. Instead, they move them around in the sentence so that they are grammatically correct for their language.

As a general rule of thumb, if you have string concatenation in your code for any user-visible strings, you should be using string formatting instead.

4.5 Choice Formatting

In some cases, you need different strings based on the number of items you have. For example, in English, you would say, “There is 1 object,” and you would say, “There are 2 objects.” The plurality of the words “is/are” and “object/objects” corresponds to the actual number of objects you are talking about. You might be tempted to solve this problem by writing code like this:

```
var str;
if (objects === 1) {
     str = resBundle.getString("There is 1 object.");
} else {
     str = resBundle.getString("There are {num} objects.");
}
```

Unfortunately, that type of code assumes English grammar. In English, there is an odd convention where one item is singular, two or more items are plural, and most bizarrely, zero items are also plural. That is, you would say, “There are 0 objects.”

In other languages, zero is not plural. The rules can get even more complex. For example, in Russian, words that indicate plurality with numbers that end with a 2 through 4 have a different ending than 5 through 9. That means the above code is not translatable to Russian!

To solve this problem, you should use the *formatChoice()* method of the *ilib.String* class. The modified code would look like this:

```
var choice = resBundle.getString(
    "1#There is 1 object.|#There are {num} objects."
);
var str = choice.formatChoice(objects, {num: objects});
```

The template string for the *formatChoice()* method has a number of parts. In this example, there are 2 choices separated by a vertical bar. The first choice is selected if the pivot number is 1 (the number before the hash character), and the resulting string should be “There is 1 object.” The second choice has no number before the hash character, so it is the default choice. In that case, the string returned is “There are {num} objects.” where “{num}” is replaced with the value of *objects* first. That is, *formatChoice()* formats the string using the values of the properties in its second argument.

Can this support Russian? Yes, because now the translator or localization engineer is free to add or remove choices in the translated string. In Russian, the translator can add a case for 0, 1, 2, 3, and 4, and make the default case be the one that supports 5 through 9. There is no

facility (yet) to specify “any number that ends in the digit 5”, but that is on the roadmap.

The syntax of the choice format string includes the ability to specify numeric ranges using “less than” or “greater than”. It also has support for string pivots, allowing you to do choices based on the value of a string. See the *ilib.String* documentation for more details.

4.6 Pseudo-Translation

The resource bundle can also help your testers discover unresourcified strings using pseudotranslation. Unresourcified strings are ones that are displayed to the user, but which have not yet been wrapped in a *getString* call.

To do this testing, you must set the default locale of ilib to the “unknown” locale, which is specified as “xx-XX”. The *ilib* namespace has a static *ilib.setLocale()* method that allows you to set the overall default locale for ilib which is used if no explicit locale is set in the parameters to a constructor or method. Example:

```
ilib.setLocale("xx-XX");
```

The “unknown” locale causes the *ResBundle* class to return strings that are pseudo-localized. That is, many of the regular Latin characters are replaced algorithmically with accented versions of those same base characters. This is still readable by an English-speaking-only QA person, and yet they immediately know that the string was resourcified properly. If a string appears in plain text, then it is either user-entered data, strings from 3rd party software or services, or it is a string that has not been resourcified properly. The QA person can submit a bug for the engineer to investigate which of those possibilities it is, and fix the code if it turns out to be a string that is not wrapped with a *getString()* method call.

Here is what a pseudo-localized string looks like:

```
"Greetings from Paris" -> "Ĝŕëëţíñğš fŕõm Pàŕíš"
```

Things get a little tricky if the string is intended to be formatted or used in a web page. In general, it is not a good idea to put HTML inside of a translatable string because the translators are linguists, not computer scientists, and they do not fully understand the syntax of HTML. However, sometimes it is necessary to do this.

For example, if you had a string like this:

```
"Greetings from <a href='url'>{city}</a> &amp; {country}"
```

Now if you just translated each character, you would end up with a string like this:

```
"Ĝŕëëţíñğš fŕõm <à ĥŕëf='úŕl'>{çíţÿ}</à> &àmþ; {çõüñţŕÿ}"
```

Obviously, that would not work as intended in the browser, as the HTML tags and entities are also pseudotranslated, as are the replacement parameters. In this case, you should instantiate your *ResBundle* with the type property set to “html”, and the psuedo-translation code will automatically parse the strings and skip any HTML tags or entities.

In the examples above, we already have the type property set correctly. With the type parameter set, the above example would come out as this instead:

```
"Ĝŕëëţíñğš fŕõm <a href='url'>{city}</a> &amp; {country}"
```

Note that the *ilib.String* replacement parameters have been left alone as well. Without that, the replacement would not work too well either!

See the iLib reference documentation for a full description of all the parameters to the *ResBundle* and *String* classes.

4.7 Formatting Numbers

The way numbers are written is locale-sensitive. In some countries, they use a comma as the decimal separator, and in some like the US, they use a period. In some countries, they use a comma as a thousands separator, in others they use a period, and yet others, they use a half space. In some countries, they even use a ten thousands separator instead of a thousands separator.

To complicate matters, currency is even more difficult. In Germany, a rounded amount of Euros might be written as “8,-- €” with dashes instead of zeros and the currency symbol written after the number. In other countries that use the Euro, the symbol comes before the number and sometimes after, sometimes with a space, sometimes without.

It is a big complicated mess! But luckily, it can easily be avoided by using the iLib number formatter class. The number formatter can format floating point numbers, currency, and percentages properly for any locale.

Let’s take a non-internationalized number formatting example and internationalize it:

```
myDiv.innerHtml =
     bundle.getString("The result is {num}.").format({
          num: someCalculation()
     });
```

Note that we have learned from the previous chapter and properly resourcified the string, and we use string formatting to interpolate the number into it in the right place before displaying it. However, the number is still formatted incorrectly for many locales.

The solution is to use the *ilib.NumFmt* class to format the number. It is very similar to date formatter in the previous chapter:

```
var fmt = new ilib.NumFmt();
myDiv.innerHtml =
     bundle.getString("The result is {num}.").format({
          num: fmt.format(someCalculation())
     });
```

Yes, it's that simple! The output from the above code when run in German and where the *someCalculation()* function returns 12345.67 would be:

```
Der Resultat ist 12.345,67.
```

That's the number 12345.67 written with German conventions.

Now, it can get more complicated of course. Let's say you want to format an amount of currency because you are selling something in Germany. Let's say for the sake of argument that we don't know what the currency for Germany is. We can let iLib figure it out for us.

```
var fmt = new ilib.NumFmt({
     type: "currency",
     locale: "de-DE"
});
myDiv.innerHtml =
     bundle.getString("This item costs {num}.").format({
          num: fmt.format(someCalculation())
     });
```

The output in the web page would look like this:

```
Dieser Artikel kostet 8,00 €.
```

The currency symbol appears after the amount, and the decimal is a comma. If the same code were run with the locale “en-IE” (English/Ireland) which also uses the Euro for its currency, the output would be:

```
This item costs €8.00.
```

The number formatter knows what the default number of fractional digits is for each type of currency, so you don’t have to specify it. You can override it, but it is probably not a good idea.

Another thing that the number formatter can do is format percentage amounts correctly for the locale. To do that, all you need to do is set the “type” property to “percentage”:

```
var fmt = new ilib.NumFmt({
     type: "percentage",
     locale: "fr-FR"
});
myDiv.innerHtml =
     bundle.getString("About {percentage} of people agree with
this.").format({
          percentage: fmt.format(someCalculation())
     });
```

This would give the output like this:

```
Environ 5 % des gens sont d'accord avec cela.
```

There are a number of other things you can do with the parameters for the number formatter constructor:

- You can specify the maximum number of digits for the fractional part of the number, and the formatter will round the number at the right place to achieve that.
- You can also specify the rounding method, which is useful for doing calculations of currency in other countries, where the generally accepted accounting practices of that

country specify a different method than is used in the US.

- You can also specify the minimum fractional digits as well. This allows you to have a number of zero digits at the end of the fractional part that are not omitted.
- When formatting currency, you can use the common style where the amount includes the currency symbol, or the ISO style where the amount includes the ISO 4217 code for the currency instead. There are many countries that use the "$" symbol for example, so if you encounter an amount like "$7.00", you are not sure if it is an amount in Mexican pesos, Australian dollars, Canadian dollars, or US dollars. That is why businesses often use the ISO style – it is unambiguous. To specify the style in the output, set the "style" property in the constructor to "common" or "iso". The common style is default.
- When formatting a currency amount with a known currency type, you can pass in the ISO 4217 code for that currency type in the "currency" property. This overrides the currency type that is default for the locale.
- When formatting a currency, you do not have to specify the number of fraction digits to show until you are overriding the default for that currency type.
- If you are formatting a regular number (not a currency or percentage amount), you can set the "style" property to "scentific" to generate a number in scientific notation, or "standard" which formats numbers according to the conventions of the locale. The standard style is default.

4.8 Locales

In internationalization-speak, a "locale" is a way of describing a set of cultural conventions. A locale is specified as a combination of a language and a region/country, and optionally also a variant.

In iLib, there is a locale class *ilib.Locale* which is instantiated using a locale specifier string. The specifier is given as a language code (the two letter ISO 3166-2 code), followed optionally by a dash, followed by a country code (the two letter ISO 639 code) for the region. Further, another dash and a variant specifier can be added to differentiate the locale even more.

Once a locale object is instantiated, it can be used to parse the specifier and give the component parts. If you want to know the language or region of a locale, you can instantiate a locale object and call its *getLanguage()* or *getRegion()* methods.

In most parts of iLib, you can pass a locale specifier instead of a locale instance to specify the locale. It is only really when you want to know the component parts of a locale that you need the locale class.

4.9 Locale Settings

There are other libraries available on the Internet that can do such things as formatting dates using a flexible formatting language like the one iLib uses. Why is iLib any different?

One important difference between a simple library of formatting and parsing routines and a true internationalization library is the ability to look up the settings and formats for a locale automatically and use them to format or parse things. That difference means that the caller does not have to be an expert on what the format is for any particular locale, and does not have to pass that format in as a parameter to the formatting method. That difference also means that all of the international-related functionality is isolated to the library, and the caller can focus on core functionality for the app or web page.

4.9.1 Locale Info

In order to retrieve information about the settings for a particular locale, you must first create an *ilib.LocaleInfo* instance for the locale:

```
var li = new ilib.LocaleInfo("it-IT");
// get the locale info for Italy
```

The locale info class has a number of different methods that are very simple, and don't really need a lot of contrived examples in order to understand them. Amongst the various functions are such routines as whether or not the locale commonly uses a 12- or 24-hour clock, what type of calendar the locale uses, the first day of the week in a calendar, the decimal separator character for numeric formats, and what type of currency the locale uses. Please see the reference documentation for the full list of methods available.

4.9.2 Currency Info

To retrieve information about a particular type of currency, you can use the *ilib.Currency* class. There are three ways to find information about currencies:

1. By locale. The currency can be given as a locale, and info about the currency most commonly used in that locale is returned.
2. By ISO code. This look-up method is the best way to get info about a particular currency because ISO codes are unambiguous.
3. By currency sign. The code finds the currency that uses that sign. If there are multiple currencies that use the same sign, the one with the largest circulation is returned. For example, the sign "$" will return US Dollars, as it is the largest of the currencies that use the dollar sign as its symbol.

Here is an example of retrieving currency information by locale:

```
var cur = new ilib.Currency({
     locale: "ja-JP"
```

```
});
console.log("Currency for ja-JP is ISO code " + cur.getCode());
// prints out "Currency for ja-JP is ISO code JPY"
```

This is an example of how to retrieve the currency information via the ISO code:

```
var cur = new ilib.Currency({
     code: "JPY"
});
console.log("Default fraction digits for JPY is " +
cur.getFractionDigits());
// prints out "Default fraction digits for JPY is 0"
```

Finally, here is how you would find the currency info by its currency sign. Note that the Japanese Yen is the only currency using the yen sign, so it returns a predictable currency.

```
var cur = new ilib.Currency({
     sign: "¥"
});
console.log("Currency that uses the ¥ symbol is " +
cur.getCode());
// prints out "Currency that uses the ¥ symbol is JPY"
```

Currently, iLib does not provide currency conversion methods.

The currency info object can provide info about the following things:

- the ISO 4217 code for the currency
- the name of the currency in English
- the default number of fraction digits
- the default currency sign

4.9.3 Time Zone Info

Time zones are notoriously difficult to deal with because politics are involved. Any time that happens, things get messy. Governments at various levels have the ability to decide which time zone their jurisdiction is in, whether or not daylight savings time (DST) is observed, and when the start and end of DST is. In the US, this even goes down to the county level!

Fortunately, ICANN publishes a set of configuration files multiple times a year that documents the latest time zone updates for all countries of the world. These files, once converted to JSON

format, can be used directly by the *ilib.TimeZone* class to give information about a particular time zone.

Time zones are specified as a region, a slash, and a large city within that region that represents the region. eg. “America/Los_Angeles” is the name of the time zone for much of the west coast of the US.

Information about a particular time zone can be looked up as a locale, a time zone specifier, or a simple offset from UTC.

Let’s say we want to look up a time zone because we want to know whether or not a particular date of the year is within DST or not.

The code is pretty simple:

```
var date = new ilib.Date.GregDate({
     year: 2012,
     month: 4,
     day: 26
});
var tz = new ilib.TimeZone({
     locale: "en-US"
});
console.log("The date April 26 is " +
     (tz.inDaylightTime(date) ? "" : "not ") +
     "in daylight savings time.");
```

Now, you may be wondering about the locale specifier for the time zone. Many countries only have one time zone, but a number are large enough to require multiple. If there are multiple time zones in a particular country, the above code will only retrieve the one that is most common and ignore the other ones. In the example above, it will return the eastern time zone, as that time zone has the largest population in the US living withing its borders.

In order to get a particular time zone unambiguously, you have to create an *ilib.TimeZone* instance by passing a specifier to its constructor in the *id* property:

```
var date = new ilib.Date.GregDate({
    year: 2012,
    month: 4,
    day: 26
});
var tz = new ilib.TimeZone({
```

```
    id: "America/Los_Angeles"
});
console.log("The date April 26 is " +
    (tz.inDaylightTime(date) ? "" : "not ") +
    "in daylight savings time.");
```

Once you have a time zone instance, you can query the following things from it:

- Get the regular offset from Universal Coordinated Time (UTC)
- Get the offset from UTC for a particular date. This can change depending on whether or not DST is in effect on that date of the year, and depending on when the politics of the day dictated when DST starts and ends in that year.
- Get the abbreviation that is used to denote the time zone on a particular date. This abbreviation can change when DST is in effect. eg. regular time on the US west coast is “PST” whereas in the summer with DST, it is “PDT”. It also depends on the year, as the start and end dates of DST change over time for each jurisdiction.
- Whether or not this time zone observes DST at all
- If this time zone does observe DST, how many hours and minutes is the change?
- The ID of this time zone

4.9.4 Calendar Info

Different countries in the world use different calendars for their dates. The number of months in a year varies, the length of months, the number of days in a month, etc.

The most accepted type of calendar in the entire world is the Gregorian calendar. However, some countries use a different calendaring system, and yet others use both the Gregorian calendar for business and another calendar for other types of events such as religious celebrations.

Ilib allows you to calculate dates in any of these calendars, convert a date from one calendar to another, and get information about the calendars.

Let’s say we have a date in the Julian calendar that we would like to convert to the Gregorian calendar.

In order to do that, we need to first understand the concept of a Julian Day. The name of Julian Days bears an unfortunate similarity to the Julian calendar, but in reality they are only related in that they a way of representing time. One is not derived from the other. A Julian Day is an interval of time in days and fractions of a day since January 1, 4713 BCE UTC at noon. Its advantage is that it does not concern itself with years and months and such. It is merely the total number of days, plus a fractional part of that day since the beginning of its epoch in 4713 BCE. In our calculations, Julian Days are used as the fixed measurement of time from which dates in

other calendars can be calculated.

Using Julian Days, it is easy to convert between one calendar and another. First, convert a date in the source calendar to its equivalent Julian Day, and then convert that Julian Day to a date in the target calendar.

Here is an example conversion of a day in the Julian calendar to its equivalent in the Gregorian:

```
var julianDate = new ilib.Date.JulDate({
    year: 2011,
    month: 8,
    day: 23
});
var gregorianDate = new ilib.Date.GregDate({
    julianday: juliandDate.getJulianDay()
});
```

4.10 Character Types

Often, when parsing a string, the code will need to know what type of character it is processing in order to decide what to do with it. Javascript does not come with any built-in functions like the C/C++ CType macros which could assist in parsing. However, iLib does.

Let's take as an example code that is commonly written in Javascript -- a function to verify the complexity of a password before a new account is created or the password is changed.

```
function checkComplexity (password) {
     var lower = 0,
          upper = 0,
          digits = 0,
          punct = 0,
          i;

     for (i = 0; i < password.length; i++) {
          if (password.charAt(i) >= 'a' &&
                    password.charAt(i) <= 'z') {
               lower++;
          } else if (password.charAt(i) >= 'A' &&
                    password.charAt(i) <= 'Z') {
               upper++;
          } else if (password.charAt(i) >= '0' &&
                    password.charAt(i) <= '9') {
```

```
                digits++;
            } else {
                if ("`!@#$%^&*()-=[]\\;\',./~_+{}|:\"<>?"
                            .indexOf(password.charAt(i)) != -1) {
                    punct++;
                }
            }
        }

        return (lower >= 1 &&
            upper >= 1 &&
            digits >= 1 &&
            punct >= 1 &&
            password.length >= 8);
}
```

Here we have 4 classes of characters that we are checking for to make sure they are present in the password: lower-case letters, upper-case letters, digits, and punctuation.

Unfortunately, someone in Russia that is used to typing their password with Cyrillic characters will be very frustrated with the above code because it excludes the lower- and upper- case Cyrillic characters and focuses exclusively on Latin characters. Here is how you can change the code above to make the above function work for any basic Unicode characters that are lower- or upper-case, no matter what script they are. (By “basic Unicode characters”, we mean any characters in the Basic Multilingual Plane of Unicode, excluding the high- and low-surrogates used for encoding UTF-16.)

```
[...]
        for (i = 0; i < password.length; i++) {
            if (ilib.CType.isLower(password.charAt(i))) {
                lower++;
            } else if (ilib.CType.isUpper(password.charAt(i))) {
                upper++;
            } else if (ilib.CType.isDigit(password.charAt(i))) {
                digits++;
            } else if (ilib.CType.isPunct(password.charAt(i))) {
                punct++;
            }
        }
[...]
```

The iLib character type functions contain information about the classes that each character

belongs to based on the Unicode 6.1 specification.

The iLib character type functions can distinguish whether a character belongs to the following character classes:

- ilib.CType.isAlnum - alphanumerics
- ilib.CType.isAlpha - alphabetics
- ilib.CType.isAscii - plain ASCII chars
- ilib.CType.isBlank - blank characters (space or tab)
- ilib.CType.isCntrl - control characters
- ilib.CType.isDigit - numeric digits
- ilib.CType.isGraph - any printable characters other than a space. (ie. not control characters)
- ilib.CType.isIdeo - ideographic characters (mostly Chinese, Japanese, Korean)
- ilib.CType.isLower - lower-case alphabetic characters
- ilib.CType.isPrint - any printable characters, including a space. (ie. not control characters)
- ilib.CType.isPunct - punctuation characters
- ilib.CType.isSpace - any whitespace characters, including space and tab
- ilib.CType.isUpper - upper-case alphabetic characters
- ilib.CType.isXdigit - hexadecimal digits (0 through 9 plus ‘a’/’A’ through ‘f’/’F’)

5.0 Assembling a Javascript File

When making an internationalized product or web site with Javascript, you might not need all of the classes and locales that iLib supports. Web sites especially need the smallest JS file possible in order to minimize the overall page load time in the browser. The iLib project includes a Java-based tool that allows you to assemble a minimized JS file that only contains the classes and locales you actually use. If the Google Closure Compiler is run on the assembly results, the size of this file becomes even smaller.

5.1 Running the Assembly Tool

In order to run the assembly tool, you will need to install a few things and make them available in your path:

- A JDK. Either OpenJDK or the standard Sun JDK will work, version 1.6 or later
- Apache ant

The tool works equally well with OpenJRE as it does with the standard Sun JRE when compiled

with the corresponding JDK. The jar file that ships with the iLib distribution is compiled with OpenJDK. Though they are supposed to be compatible, the reality is that sometimes the binary files are not. If you would like to use the Sun JDK instead, make sure you have Apache ant installed and both ant and the Sun JDK in your path, and then issue the command “ant clean dist” in the root of iLib in order to recompile the code. This will avoid any compatibility issues.

5.2 Command-line Arguments

The JS Assembler tool can be run on the command-line using the “jsa” script in Unix/MacOS, or “jsa.bat” on Windows. These scripts simply call the JRE to run the tool. You may have to edit these scripts to change where your files are stored.

The tool can also be called from an ant script by invoking the JRE, though no direct ant task has been created yet. The build.xml file that comes with iLib assembles the iliball.js and iliball-compiled.js files. You can modify it to build only those parts you need.

The tool has a number of command-line options:

```
Usage: jsa [-i include_dir] [-o outfile] [-l locale_list] [js_file_name ...]

-i include_dir      - add an include directory to the path to find other
                      js files.
-o outfile          - specify the resulting assembled output file. Default is
                      to send the output to stdout.
-l locale_list      - specify a comma-separated list of locales to use when
including data files.
js_file_name        - name of a Javascript file to process. If none given, the
                      current directory is recursively searched for all
                      Javascript files.
```

The assembly tool reads the named javascript files, looking for dependencies. It then reads each of the javascript files that are depended upon, and does the same thing recursively, until it can find no more unsatisfied dependencies. Then, it does a topological sort to order all the dependencies, and creates an output file with all the dependencies listed in the correct order such that nothing is used before it is defined and each file is only included once.

The -i include directory parameter allows you to keep libraries of Javascript code in different directories, and depend on them without needing to explicitly name their whole paths. This is similar to the way the C preprocessor works with #include directives.

5.3 Specifying Dependencies

Dependencies are specified with a depends directive which has the following syntax:

```
// !depends filename.js [filename.js …]
```

```
or

/* !depends filename.js [filename.js …] */
```

That is, they are encoded within comments in the Javascript so that the Javascript engine can ignore them when the code is run. Anything in the same comment block is considered to be a file name, so make sure to make these directives live in their own separate comment block.

5.4 Specifying Locale Data

For the most part, code that uses iLib will not need to specify the locale data directly and this section can be skipped by most readers. The JS files that your code depends upon will specify the dependencies on their own locale data.

However, if you are using the assembly tool with your own Javascript sources, you may need to know how it works so that you can include your own locale-dependent data.

Locale data is data that configures how the general locale-sensitive functions operate for a specific locale. For example, the date formatting data configures what the date formats are for a particular locale.

The data for a particular locale are only included in the assembled file when the locale is specified on the command-line for the assembly tool. Make sure you get the correct list of locale when you invoke the assembly tool or else the code will not function as expected.

A data dependency is specified with a data directive similar to the depends directive:

```
// !data datatype [datatype …]

or

/* !depends datatype [datatype …] */
```

Note that the names given for the data directive are not file names as in the depends directive, and there are no file name extensions either. The reason is that the file name is constructed out of the datatype name. All locale-dependent data is specified in json files so the file name extension is always “.json”. The locale-dependent data is loaded from a directory tree named after the locales, where the first level is the language name, and the 2nd level underneath there is the name of the region/country. For example, the date formats for US English would be specified as “!data dateformats” in the directive, and would be loaded from the file

en/US/dateformats.json.

Locale-dependent files are cascaded in code similar to the way CSS style sheets are cascaded in a browser. The most generic data outside of the locale directories is loaded first, followed by the language-specific data, followed by the data that is both language- and region-specific. Any settings that appear at more generic levels are overridden by the data in the more specific levels. For example, this allows you to specify settings for English such as the measurement system set to “metric”, seeing as most English-speaking locales use the metric system. Then, you can override that setting just for en-US and set it to “uscustomary”.

Some locale-data, such as the measurement system commonly used in that locale, is very small. The measurement system settings for example, consist of just one property with a string value of “metric”, “uscustomary” or “imperial”. That’s it. There are a number of other settings like this as well. Rather than having a large proliferation of very small objects that are optionally included in the output, these smaller settings are lumped together in one file object called a LocaleInfo. You can retrieve the settings easily using the ilib.LocaleInfo class.

5.5 Meta-files

One problem with the assembly tool right now is that all files specified on the command-line are included in the output file, which is sometimes not the behaviour you are looking for, especially if the files specified on the command-line are HTML files rather than independent Javascript files.

In order to assemble only the Javascript files you need without including your own sources, you can create a meta-file whose only contents are a comment with the depends directive in it. The iliball.js file was constructed this way, and you can look at js/src/ilib.js for its meta-file. You can either modify this file and then run “ant dist” to assemble a new copy of iliball.js, or you can create your own meta-file.

There are future plans to give the assembly tool the ability to assemble a file without also including the files that depend on iLib. That is, give the tool the ability to read an HTML file for depends statements and only assemble an output file of Javascript files that the HTML file depends upon. Until then, use the meta-file approach.

5.6 Roll Your Own

The assembly tool is not specific to iLib. You can use it to assemble your own JS files for your own web site. Use the depends and data directives in your own Javascript files, and you can create one properly sequenced file that includes your own classes and the iLib classes all together in one short file.