Localized Number Formatting in ICU and Beyond

Internationalization & Unicode Conference 41,

October 17, 2017

Presented by Shane Carr,

Software Engineer,

Internationalization, Google

Brief History of Numbering Systems

Prehistoric Methods for Counting

Bag of Pebbles: each pebble corresponds to one sheep in the flock.

Knots: The number of knots tied in a rope counts the number of items.

Tally Marks: Still used today when counting by hand.

Language: Earliest languages included words for small quantities but were not able to count large quantities.

A stone some archaeologists believe to be engraved with tally marks, dated to at least 70,000 years old.
(Chip Clark, Smithsonian Institution)

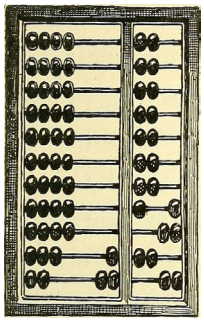
http://humanorigins.si.edu/evidence/behavior/recording-information/blombos-ocher-plaque

First Functional Numeral Systems

Egyptian (3000 BCE): Base 10 with hieroglyphics for one, ten, hundred, thousand, etc. You write as many as you need to sum to your quantity.

Babylonian (2100 BCE): Base 60 with groups of numerals representing the number of ones, 60s, 3600s, and so on.

Mayan (ca. 300 CE): Base 20 positional system; first system with a zero digit.



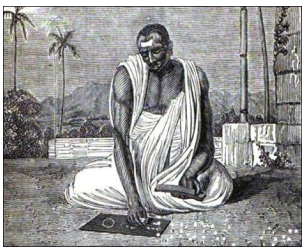
An abacus was used for performing calculations before the development of the positional decimal numeral system. (1911 textbook illustration, https://flic.kr/p/owaP8C)

Indo-Arabic Numeral System

Developed in India around 500 CE.

Base 10 "positional" numbering system: unique symbols for 0 through 9, with positions corresponding to powers of ten.

Adopted by the Arabs and then by the Europeans. The printing press established Indo-Arabic as the dominant numeral system in the West.



Brahmagupta, a *dybuck* (Indian astronomer), is credited with inventing the concept of zero. Shown here is an 1885 illustration of a dybuck.

https://books.google.ch/books?id=85UDAAAAQAAJ&pg=PA318

Numbers in Unicode

Scripts and Numbering Systems

Most scripts throughout the world include the ten Indo-Arabic numerals. Some scripts include two or more variants.

Unicode calls each set of numerals a "numbering system".

Indo-Arabic Numbering Systems

ASCII-range Indo-Arabic numerals in the Latin, or *latn*, script

"latn"	0123456789
"mathbold"	𝟎𝟏𝟐𝟑𝟒𝟓𝟔𝟕𝟖𝟗
"mathdbl"	𝟘𝟙𝟚𝟛𝟜𝟝𝟞𝟟𝟠𝟡
"mathmono"	𝟶𝟷𝟸𝟹𝟺𝟻𝟼𝟽𝟾𝟿
"mathsanb"	𝟬𝟭𝟮𝟯𝟰𝟱𝟲𝟳𝟴𝟵
"mathsans"	𝟢𝟣𝟤𝟥𝟦𝟧𝟨𝟩𝟪𝟫
"bali"	᭐᭑᭒᭓᭔᭕᭖᭗᭘᭙
"cham"	꩐꩑꩒꩓꩔꩕꩖꩗꩘꩙
"brah"	𑁦𑁧𑁨𑁩𑁪𑁫𑁬𑁭𑁮𑁯

"deva"	०१२३४५६७८९
"gujr"	૦૧૨૩૪૫૬૭૮૯
"guru"	੦੧੨੩੪੫੬੭੮੯
"lepc"	᱀᱁᱂᱃᱄᱅᱆᱇᱈᱉
"tibt"	༠༡༢༣༤༥༦༧༨༩
"orya"	୦୧୨୩୪୫୬୭୮୯
"saur"	꣐꣑꣒꣓꣔꣕꣖꣗꣘꣙
"beng"	০১২৩৪৫৬৭৮৯
"khmr"	០១២៣៤៥៦៧៨៩
"mong"	᠐᠑᠒᠓᠔᠕᠖᠗᠘᠙

Most numbering systems are named after their ISO 15924 script code.

Indo-Arabic Numbering Systems

Native Arabic numerals: Indo-Arabic numerals in the Arabic script

"arab"	٠١٢٣٤٥٦٧٨٩
"arabext"	۰۱۲۳۴۵۶۷۸۹

"java"	꧐꧑꧒꧓꧔꧕꧖꧗꧘꧙
"kali"	꤀꤁꤂꤃꤄꤅꤆꤇꤈꤉
"knda"	೦೧೨೩೪೫೬೭೮೯
"lanatham"	᪀᪁᪂᪃᪄᪅᪆᪇᪈᪉
"laoo"	໐໑໒໓໔໕໖໗໘໙
"limb"	᥆᥇ᥒᥓᥔᥕᥖᥗᥘᥙ
"mlym"	൦൧൨൩൪൫൬൭൮൯
"mtei"	꯰꯱꯲꯳꯴꯵꯶꯷꯸꯹
"talu"	᧐᧑᧒᧓᧔᧕᧖᧗᧘᧙
"lana"	᪀᪁᪂᪃᪄᪅᪆᪇᪈᪉
"mymr"	၀၁၂၃၄၅၆၇၈၉

"mymrshan"	႐႑႒႓႔႕႖႗႘႙
"nkoo"	߀߁߂߃߄߅߆߇߈߉
"olck"	᱐᱑᱒᱓᱔᱕᱖᱗᱘᱙
"osma"	𐒠𐒡𐒢𐒣𐒤𐒥𐒦𐒧𐒨𐒩
"sund"	᮰᮱᮲᮳᮴᮵᮶᮷᮸᮹
"telu"	౦౧౨౩౪౫౬౭౮౯
"thai"	๐๑๒๓๔๕๖๗๘๙
"vaii"	꘠꘡꘢꘣꘤꘥꘦꘧꘨꘩

"tamldec" ௦௧௨௩௪௫௬௭௮௯

And more! (Sorry if I didn't include your favorite)

Facts about Indo-Arabic numerals

1. All numerals have Unicode properties including a numeric value
2. Contiguous: digit 0 at code point *x* → digit 1 at code point *x*+1
3. Usually in same encoding block as corresponding script
4. Most significant digit usually on the left
   a. Including Arabic, an otherwise right-to-left script
   b. Some exceptions (e.g., Adlam)
5. Majority of CLDR locales use Latin-script digits as default
6. Not necessarily in the basic multilingual plane

Algorithmic Numbering Systems

Unicode calls all non-Indo-Arabic numbering systems "algorithmic." These numbering systems are sometimes used in formal or financial contexts. Examples:

Unicode Name	39	500	2017
roman	XXXIX	D	MMXVII
hant	三十九	五百	二千零一十七
cyrl	л҃ѳ	ф҃	҂вз҃і
taml	௩௰௯	௫௱	௨௲௰௭

Others include: Armenian, Ethiopic, Greek, Georgian, Hebrew, Japanese, and variants.

Locale Patterns and Symbols

Different characters have different semantic meanings by locale.

Check the CLDR Charts!

http://www.unicode.org/cldr/charts/latest/

, , ·

Number Formatting in ICU

Two aspects of number processing on a computer:

Formatting: binary to human-readable

Today's Talk

Parsing: human-readable to binary

Two APIs for Number Formatting

DecimalFormat

Dates to the original libraries from Taligent and IBM in the mid-1990s.
A foundational i18n API, laying the groundwork JDK, EcmaScript, and others.

NumberFormatter

The first major overhaul of number formatting APIs in ICU.
Newly released in ICU 60.

Today, I will introduce NumberFormatter.

Real DecimalFormat Call Site (Java)

The need for a cast is ugly

```
static DecimalFormat percent = (DecimalFormat)
    NumberFormat.getPercentInstance();
static {
    percent.setMaximumFractionDigits(6);
    percent.setPositivePrefix("+");
}
```

The locale is defaulted and fixed to the Java system locale, which is not normally recommended (except on Android); also encourages the inefficient "create & destroy" pattern

Locale-unaware plus sign code point and position

Real DecimalFormat Call Site (C++)

A lot of code to create a formatter that uses currency symbols to display a number to 6 decimal places, which should be simple

```
auto* format = dynamic_cast<icu::DecimalFormat*>
 (icu::NumberFormat::createCurrencyInstance(error));
format->setRoundingIncrement(0.0);
auto symbols = *(format->
    getDecimalFormatSymbols());
symbols.setSymbol(
    icu::DecimalFormatSymbols::kCurrencySymbol, "");
format->setDecimalFormatSymbols(symbols);
format->applyPattern("#,##0.000000", error);
```

What does this mean? (Clunky way to say that you don't want currency increment rounding rules)

Locale-unaware grouping size

Heap allocation overhead getting and setting DecimalFormatSymbols

In just two call sites, that's a lot of issues!

Can we do better?

What inspired NumberFormatter?

DecimalFormat's design has limitations which have become more apparent over the last 20 years of advances in language design and demands for number formatting.

- Not designed for multicore architectures
- Difficult to specify certain options in a locale-agnostic way
  - Methods such as setPositivePrefix() and setGroupingSize() are intrinsically locale-dependent
  - A new object is required for every locale, particularly problematic on servers
- Awkward behaviors: can't be fixed because of backwards compatibility
- API clutter: over 30 settings, many of which overlap or are obsolete
- ICU4C depends on heap allocation and does not take advantage of language and compiler advances
- Formatting and parsing are intertwined, when in practice the needs are different

NumberFormatter Design

Well over 100 call sites of DecimalFormat were analyzed to see how programmers interacted with the old API. This led to the following goals:

Locale: Settings should be locale-agnostic, so you can choose to specify your locale during application startup (good on devices) or at the final call site (good on servers).

Orthogonality: NumberFormatter settings should be *orthogonal* to the greatest extent possible: the choice on one setting should not affect the behavior of other settings.

Thread Safety: All objects should be immutable and thread-safe. Settings could be given in a "fluent chain," a design pattern popularized by Google Guava.

Fluent Pattern

Settings are chained, and each intermediate element in the chain is a functional, immutable formatter object. Easy to use and no thread safety problems.

Java

```
NumberFormatter.with()  // => UnlocalizedNumberFormatter
    .settingA(...)      // => UnlocalizedNumberFormatter
    .settingB(...)      // => UnlocalizedNumberFormatter
    .locale(...)        // => LocalizedNumberFormatter
    .settingC(...)      // => LocalizedNumberFormatter
    .format(...)        // => FormattedNumber
    .toString();        // => String
```

C++

```
NumberFormatter::with() // => UnlocalizedNumberFormatter
    .settingA(...)      // => UnlocalizedNumberFormatter
    .locale(...)        // => LocalizedNumberFormatter
    .format(..., ec)    // => FormattedNumber
    .toString();        // => UnicodeString
```

Note: everything returns by value! (uses C++ return-value optimization)

Device vs. Server Usage

Device Pattern

```
private static final LocalizedNumberFormatter formatter =
    NumberFormatter.withLocale(Locale.getDefault())
        .settingA(...)
        .settingB(...);

// Call site:
formatter.format(...).toString();
```

Server Pattern

```
private static final UnlocalizedNumberFormatter formatter =
    NumberFormatter.with()
        .settingA(...)
        .settingB(...);

// Call site:
formatter.locale(...).format(...).toString();
```

Setting 1: Notation

Options:

- Scientific
- Engineering
- Compact (Short)
- Compact (Long)
- Simple

Future Possibilities:

- Spell-out / algorithmic (#13401)
- Range-dependent notation (#13403)

```
NumberFormatter.with()
.notation(Notation.compactShort())
.locale(new ULocale("ru"))
.format(-980651.4237)
.toString();
```

-981 тыс.

"Notation" not needed if compactShort() is static-imported

Notice new default rounding strategy for compact notation (affects old API!)

Setting 2: Rounding

Options:

- Fraction length
- Significant digits/figures
- Increment
- Currency rounding
- Unlimited precision (no rounding)

Note: this setting alone accounts for 9 overlapping getters and setters in DecimalFormat

```
NumberFormatter.with()
.rounding(Rounder.fixedFraction(2))
.locale(new ULocale("ru"))
.format(-980651.4237)
.toString();
```

-980 651,42

"Rounder" not needed if fixedFraction() is static-imported

Setting 3: Unit

Options:

- Percent/Permille
- Currency
- Measure unit
- None

Note: In DecimalFormat, you pick a "style", which mixes notation with unit and prevents certain combinations like scientific with percent.

```
NumberFormatter.with()
.unit(NoUnit.percent())
.locale(new ULocale("ru"))
.format(-980651.4237)
.toString();
```

-980 651,4237 %

"NoUnit" not needed if percent() is static-imported

For consistency with other units, no multiplying by 100 (old API still multiplies)

Setting 4: Integer Width

Options:

- Zero-Fill To
  - i.e., "minimum integer digits"
- Truncate At
  - i.e., "maximum integer digits"

Can be used to render numbers at a fixed width. Included because this feature was somewhat popular with current users.

```
NumberFormatter.with()
.integerWidth(
    IntegerWidth.zeroFillTo(4))
.locale(new ULocale("bn"))
.format(9.806514237)
.toString();
```

০,০০৯.৮০৬৫১৪

"IntegerWidth" not needed if zeroFillTo() is static-imported

Note: default rounding is 6 fraction places, consistent with the C standard for *printf*

Setting 5: Symbols

Options:

- DecimalFormatSymbols
- NumberingSystem

DecimalFormatSymbols is a wrapper over NumberingSystem that adds additional locale data, so it makes sense to put these into one setter.

```
NumberFormatter.with()
.symbols(NumberingSystem.LATIN)
.locale(new ULocale("bn"))
.format(9806514.237)
.toString();
```

98,06,514.237

"NumberingSystem" not needed if LATIN is static-imported

Locale still affects grouping size and other parts of the pattern

Setting 6: Unit Width

Options:

- Narrow
- Short (default)
- Full Name
- ISO Code
- Hidden

Naming is as consistent as possible with CLDR.

```
NumberFormatter.with()
.unit(Currency.getInstance("JPY"))
.unitWidth(UnitWidth.FULL_NAME)
.locale(new ULocale("ar"))
.format(980.6514237)
.toString();
```

١٩٨٠ ين ياباني

"UnitWidth" not needed if FULL_NAME is static-imported

Currency rounding used by default, but can be easily overridden by the rounding() setter

Setting 7: Sign Display

Options:

- Automatic
- Always Shown
- Never Shown
- Accounting
- Accounting-Always

```
NumberFormatter.with()
.sign(SignDisplay.ALWAYS)
.locale(new ULocale("ne"))
.format(980651.4237)
.toString();
```

+९८०,६५१.४२३७

Sign is localized if necessary and put in the correct position (before/after number)

Setting 8: Decimal Separator Display

Options:

- Automatic
- Always Shown

Affects numbers without a fraction part.

```
NumberFormatter.with()
.decimal(
    DecimalSeparatorDisplay.ALWAYS)
.locale(new ULocale("dz"))
.format(9806514237L)
.toString();
```

৯,৮০,৬৫,১৫,২৩৯.

Future Settings

Didn't make it into 60 but may be added in 61 or later:

- Grouping strategy (technical preview in 60)
  - How to control locale-sensitive minimum grouping digits?
- Padding
  - Integer width covers the biggest use case
  - Some users may want DecimalFormat-style padding with a custom character
- Range formatting
  - "1-3 meters"
  - "~5%"
- Other suggestions? File a ticket on ICU Trac.

FormattedNumber?

Calling *.format()* returns FormattedNumber, which has the following methods:

- toString()
- appendTo(appendable)
- populateFieldPosition(fp[, status])
- getFieldIterator() -- Java
- populateFieldPositionIterator(fpi, status) -- C++
- toBigDecimal() -- Java

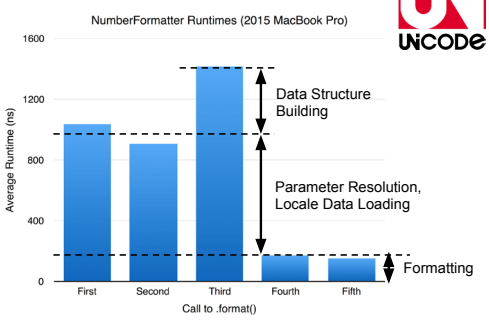
NumberFormatter is Self-Regulating

On a NumberFormatter:

- **First/Second .format():** Regular code path.
- **Third .format():** Slow code path, building structures.
- **Fourth+ .format():** Fast code path, using the structures built in call 3.

Reproduce these results:

https://pastebin.com/6V6EiXbe



Live Demo

https://goo.gl/2N2Xcq

FAQ

Q: If my old code uses DecimalFormat, do I need to update it?

A: DecimalFormat is still here and has partly become a wrapper over NumberFormatter. The API is intended for new code and as another option when refactoring old code.

Q: What about Parsing?

A: This API is focused on formatting. In an upcoming release, we may propose a separate "NumberParser" API for parse users.

FAQ

Q: I don't use Java or C++; can I still use NumberFormatter?

A: There are wrappers over ICU in all major languages, including Python, C#, PHP, JavaScript, and others. As soon as those package are updated, you should have access to NumberFormatter.

Q: I want more information or have a suggestion; where can I ask?

A: Open a ticket on ICU Trac. http://bugs.icu-project.org/trac/newticket Also consider reading the design doc for NumberFormatter, which goes into much more depth on many of the issues in this presentation. http://goo.gl/GyyF2s

Questions?

You can contact me at http://shane.guru