A Critical Analysis of String APIs: the Case of Pharo

Damien Pollet[a], Stéphane Ducasse[a]

[a]*RMoD — Inria & Université Lille, France*

Abstract

Most programming languages, besides C, provide a native abstraction for character strings, but string APIs vary widely in size, expressiveness, and subjective convenience across languages. In Pharo, while at first glance the API of the String class seems rich, it often feels cumbersome in practice; to improve its usability, we faced the challenge of assessing its design. However, we found hardly any guideline about design forces and how they structure the design space, and no comprehensive analysis of the expected string operations and their different variations. In this article, we first analyse the Pharo 4 String library, then contrast it with its Haskell, Java, Python, Ruby, and Rust counterparts. We harvest criteria to describe a string API, and reflect on features and design tensions. This analysis should help language designers in understanding the design space of strings, and will serve as a basis for a future redesign of the string library in Pharo.

Keywords: Strings, API, Library, Design, Style

1. Introduction

While strings are among the basic types available in most programming languages, we are not aware of design guidelines, nor of a systematic, structured analysis of the string API design space in the literature. Instead, features tend to accrete through ad-hoc extension mechanisms, without the desirable coherence. However, the set of characteristics that good APIs exhibit is generally accepted [1]; a good API:

- is easy to learn and memorize,
- leads to reusable code,
- is hard to misuse,
- is easy to extend,
- is complete.

To evolve an understandable API, the maintainer should assess it against these goals. Note that while orthogonality, regularity and consistency are omitted, they arise from the ease to learn and extend the existing set of operations. In the case of strings, however, these characteristics are particularly hard to reach, due to the following design constraints.

For a single data type, strings tend to have a large API: in Ruby, the String class provides more than 100 methods, in Java more than 60, and Python's str around 40. In Pharo[1], the String class alone understands 319 distinct messages, not counting inherited methods. While a large API is not always a problem *per se*, it shows that strings have many use cases, from concatenation and printing to search-and-replace, parsing, natural or domain-specific languages. Unfortunately, strings are often abused to eschew proper modeling of structured data, resulting in inadequate serialized representations which encourage a procedural code style[2]. This problem is further compounded by overlapping design tensions:

Mutability: Strings as values, or as mutable sequences.

Abstraction: Access high-level contents (words, lines, patterns), as opposed to representation (indices in a sequence of characters, or even bytes and encodings).

Orthogonality: Combining variations of abstract operations; for instance, substituting one/several/all occurrences corresponding to an index/character/sequence/pattern, in a case-sensitive/insensitive way.

In previous work, empirical studies focused on detecting non-obvious usability issues with APIs [2–4]; for practical advice on how to design better APIs, other works cite guideline inventories built from experience [5, 6]. Joshua Bloch's talk [7] lists a number of interesting rules of thumb, but it does not really bridge the gap between abstract methodological advice (e.g. *API design is an art, not a science*) and well-known best practices (e.g. *Avoid long parameter lists*). Besides the examples set by particular implementations in existing languages like Ruby, Python, or Icon [8], and to the best of our knowledge, we are not aware of string-specific analyses of existing APIs or libraries and their structuring principles.

Email addresses: `damien.pollet@inria.fr` (Damien Pollet), `stephane.ducasse@inria.fr` (Stéphane Ducasse)

[1]Numbers from Pharo 4, but the situation in Pharo 3 is very similar.

[2]Much like with Anemic Domain Models, except the string API is complex: http://www.martinfowler.com/bliki/AnemicDomainModel.html

In this paper, we are not in a position to make definitive, normative design recommendations for a string library; instead, we adopt a descriptive approach and survey the design space to spark discussion around its complexity and towards more understandable, reusable, and robust APIs. To this end, we study the string libraries of a selection of programming languages, most object-oriented for a comparison basis with Pharo, with Haskell and Rust thrown in for some contrast due to their strong design intents. We consider these languages to be general purpose and high-level enough that readability, expressivity, and usability are common goals. However, a caveat: each language comes with its own culture, priorities, and compromises; we thus have to keep a critical eye and put our findings in the perspective both of the design intent of the studied language, and of our own goals in Pharo. Similarly, we focus the study on the API of the String class or its equivalent only, and we limit the discussion of related abstractions to their interactions in the string API. Extending the study to the APIs of other text processing abstractions like streams, regular expressions, or parser combinators at the same level of detail as strings would only make the paper longer.

Section 2 shows the problems we face using the current Pharo 4 string library. In Sections 3 and 4, we identify idioms and smells among the methods provided by Pharo's String class. Section 5 examines the relevant parts of the ANSI Smalltalk standard. We survey the features expected of a String API in Section 6, then existing implementations in several general-purpose languages such as Java, Haskell, Python, Ruby, and Rust in Section 7. Finally, we highlight a few design concerns and takeaways in Section 8, before concluding the paper.

2. Pharo: Symptoms of Organic API Growth

As an open-source programming environment whose development branched off from Squeak, Pharo inherits many design decisions from the original Smalltalk-80 library. However, since the 1980's, that library has grown, and its technical constraints have evolved. In particular, since Squeak historically focused more on creative and didactic experimentation than software engineering and industrial use, the library has evolved organically more than it was deliberately curated towards a simple and coherent design.

Even though we restrict the scope of the analysis to the String class, we face several challenges to identify recurring structures and idioms among its methods, and to understand and classify the underlying design decisions.

Large number of responsibilities. As explained in Section 1, strings propose a wide, complex range of features. For example, Pharo's String defines a dozen class variables for character and encoding properties.

Large number of methods. The current Pharo String class alone has 319 methods, excluding inherited methods. However, Pharo supports open-classes: a package can define *extension methods* on classes that belong to another package [9, 10]; we therefore exclude extension methods, since they are not part of the core behavior of strings. Still, this leaves 180 methods defined in the package of String. That large number of methods makes it difficult to explore the code, check for redundancies, or ensure completeness of idioms.

Using the code browser, the developer can group the methods of a class into protocols. However, since a method can only belong to one protocol, the resulting classification is not always helpful to the user. For example, it is difficult to know at first sight if a method is related to character case, because there is no dedicated protocol; instead, the case conversion methods are all part of a larger *converting* protocol which bundles conversions to non-string types, representation or encoding conversions, extracting or adding prefixes.

Multiple intertwined behaviors. Strings provide a complex set of operations for which it is difficult to identify a simple taxonomy. Consider the interaction between features: a single operation can be applied to one or multiple elements or the whole string, and can use or return an index, an element, a subset or a subsequence of elements:

Operations: insertion, removal, substitution, concatenation or splitting

Scope: element, pattern occurrence, anchored subsequence

Positions: explicit indices, intervals, matching queries

Occurrences: first, last, all, starting from a given one

In Pharo we can replace all occurrences of one character by another one using the replaceAll:with: inherited from SequenceableCollection, or all occurrences of one character by a subsequence (copyReplaceAll:with:). Like these two messages, some operations will copy the receiver, and some other will change it in place. This highlights that strings are really mutable collections of characters, rather than pieces of text, and that changing the size of the string requires to copy it. Finally, replacing only one occurrence is yet another cumbersome message (using replaceFrom:to:with:startingAt:).

```
'aaca' replaceAll: $a with: $b                          → 'bbcb'
'aaca' copyReplaceAll: 'a' with: 'bz'                → 'bzbzcbz'
'aaca' replaceFrom: 2 to: 3 with: 'bxyz' startingAt: 2    → 'axya'
```

Lack of coherence and completeness. Besides its inherent complexity, intertwining of behaviors means that, despite the large number of methods, there is still no guarantee that all useful combinations are provided. Some features are surprisingly absent or unexploited from the basic String class. For instance, string splitting and regular expressions, which are core features in Ruby or Python, have long been third-party extensions in Pharo. They were only recently integrated, so some methods like lines, substrings:, or findTokens: still rely on ad-hoc implementations. This reveals refactoring opportunities towards better composition of independent parts.

Moreover, some methods with related behavior and similar names constrain their arguments differently. For instance, findTokens: expects a collection of delimiter characters, but also accepts a single character; however, findTokens:keep: lacks that special case. Perhaps more confusingly, some methods with similar behavior use dissimilar wording: compare the predicates isAllDigits and onlyLetters, or the conversion methods asUppercase and asLowercase but withFirstCharacterDownshifted.

Impact of immutability. In some languages such as Java and Python, strings are immutable objects, and their API is designed accordingly. In Smalltalk, strings historically belong in the collections hierarchy, and therefore are mutable.

In practice, many methods produce a modified copy of their receiver to avoid modifying it in place, but either there is no immediate way to know, or the distinction is made by explicit naming. For instance, replaceAll:with: works in-place, while copyReplaceAll:with: does not change its receiver. Moreover, the VisualWorks implementation supports object immutability, which poses the question of how well the historic API works in the presence of immutable strings.

Duplicated or irrelevant code. A few methods exhibit code duplication that should be factored out. For instance, withBlanksCondensed and withSeparatorsCompacted both deal with repeated whitespace, and findTokens: and findTokens:keep: closely duplicate their search algorithm.

Similarly, some methods have no senders in the base image, or provide ad-hoc behavior of dubious utility. For instance, the method comment of findWordStart:startingAt: mentions "HyperCard style searching" and implements a particular pattern match that is subsumed by a simple regular expression.

3. Recurring Patterns

We list here the most prominent patterns or idioms we found among the analyzed methods. Although these patterns are not followed systematically, many of them are actually known idioms that apply to general Smalltalk code, and are clearly related to the ones described by Kent Beck [5]. This list is meant more as a support for discussion than a series of precepts to follow.

Layers of convenience. One of the clearest instances in this study is the group of methods for trimming (Figure 1). Trimming a string is removing unwanted characters (usually whitespace) from one or both of its extremities.

The library provides a single canonical implementation that requires two predicates to identify characters to trim at each end

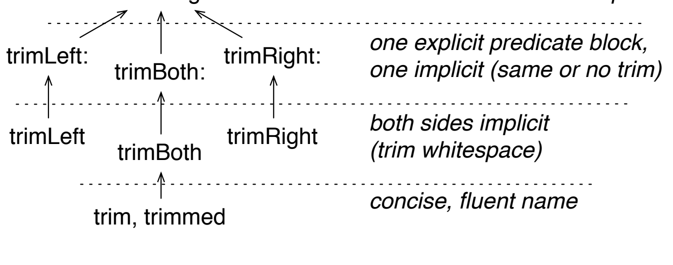
trimLeft:right:
canonical: both sides explicit
trimLeft:
trimBoth:
trimRight:
one explicit predicate block, one implicit (same or no trim)
trimLeft
trimBoth
trimRight
both sides implicit (trim whitespace)
trim, trimmed
concise, fluent name

Figure 1: Chains of convenience methods delegating to a single canonical behavior: trimming at one or both ends.

of the string. A first layer of convenience methods eliminates the need for two explicit predicates, either by passing the same one for both ends, or by passing one that disables trimming at one end (trimBoth:, trimLeft:, and trimRight:). A second layer of convenience methods passes the default predicate that trims whitespace (trimLeft, trimBoth, and trimRight). Finally, two additional methods provide concise verbs for the most common case: whitespace, both ends (trim and trimmed, which are synonymous despite the naming).

Convenience methods can also change the result type; the following list shows a few examples of convenience predicates wrapping indexing methods.

Trimming ends trim, trimmed, trimLeft:right:, trimBoth, trimBoth:, trimLeft, trimleft:, trimRight, trimRight:

Index of character indexOf:, indexOf:startingAt:, indexOf:startingAt:ifAbsent:

Index of substring findString:, findString:startingAt:, findString:startingAt:caseSensitive:, and related predicates includesSubstring:, includesSubstring:caseSensitive:

Macro expansion expandMacros, expandMacrosWith: etc., expandMacrosWithArguments:

Sort order compare:, compare:caseSensitive:, compare:with:collated:, and predicates sameAs:, caseInsensitiveLessOrEqual:, and caseSensitiveLessOrEqual:

Spelling correction correctAgainst:, correctAgainst:continuedFrom:, correctAgainstDictionary:continuedFrom:, correctAgainstEnumerator:continuedFrom:

Lines lines, lineCount, lineNumber:, lineCorrespondingToIndex:, linesDo:, lineIndicesDo:

Missed opportunity substrings does not delegate to substrings:

This idiom allows concise code when there is a convention or an appropriate default, without giving up control in other cases. However, its induced complexity depends on the argument combinations necessary; it then becomes difficult to check all related methods for consistency and completeness.

We propose to broaden and clarify the use of this idiom wherever possible, as it is an indicator of how flexible the canonical methods are, and promotes well-factored convenience methods. There are several missed opportunities for applying this idiom in String: for instance copyFrom:to: could have copyFrom: (up to the end) and copyTo: (from the start) convenience methods.

Pluggable sentinel case. When iterating over a collection, it is common for the canonical method to expect a block to evaluate for degenerate cases. This leads to methods that are more akin to control flow, and that let the caller define domain computation in a more general and flexible way.

Methods that follow this idiom typically include either ifNone: or ifAbsent: in their selector. For context, in a typical Pharo image as a whole, there are 47 instances of the ifNone: pattern, and 266 instances of ifAbsent:.

Index lookup indexOf:startingAt:ifAbsent:, indexOfSubCollection:startingAt:ifAbsent:

We promote this idiom in all cases where there isn't a clear-cut choice of how to react to degenerate cases. Indeed, forcing either a sentinel value, a Null Object [11], or an exception on user code forces it to check the result value or catch the exception, then branch to handle special cases. Instead, by hiding the check, the pluggable sentinel case enables a more confident, direct coding style. Of course, it is always possible to fall back to either a sentinel, null, or exception, via convenience methods.

Sentinel index value. When they fail, many index lookup methods return an out-of-bounds index; methods like copyFrom:to: handle these sentinel values gracefully. However, indices resulting from a lookup have two possible conflicting interpretations: either *place of the last match* or *last place examined*. In the former case, a failed lookup should return zero (since Smalltalk indices are one-based); in the latter case, one past the last valid index signifies that the whole string has been examined. Unfortunately, both versions coexist:

```
'abc' findString: 'x' startingAt: 1                 → 0
'abc' findAnySubStr: #('x' 'y') startingAt: 1       → 4
```

We thus prefer the pluggable sentinel, leaving the choice to user code, possibly via convenience methods.

Zero index findSubstring:in:startingAt:matchTable:, findLastOccurrenceOfString:startingAt:, findWordStart:startingAt:, indexOf:startingAt:, indexOfFirstUppercaseCharacter, indexOfWideCharacterFrom:to:, lastSpacePosition, indexOfSubCollection:

Past the end findAnySubStr:startingAt:, findCloseParenthesisFor:, findDelimiters:startingAt:

Iteration or collection. Some methods generate a number of separate results, accumulating and returning them as a collection. This results in allocating and building an intermediate collection, which is often unnecessary since the calling code needs to iterate them immediately. A more general approach is to factor out the iteration as a separate method, and to accumulate the results as a special case only. A nice example is the group of line-related methods that rely on lineIndicesDo:; some even flatten the result to a single value rather than a collection.

Collection lines, allRangesOfSubstring:, findTokens:, findTokens:keep:, findTokens:escapedBy:, substrings, substrings:

Iteration linesDo:, lineIndicesDo:

In our opinion, this idiom reveals a wider problem with Smalltalk's iteration methods in general, which do not decouple the iteration per se from the choice of result to build — in fact, collections define a few optimized methods like select:thenCollect: to avoid allocating an intermediate collection. There are many different approaches dealing with abstraction and composeability in the domain of iteration: push or pull values, internal or external iteration, generators, and more recently transducers [12, 13].

Conversion or manipulation. String provides 24 methods whose selector follows the as*Something* naming idiom, indicating a change of representation of the value. Conversely, past participle selectors, e.g. negated for numbers, denote a transformation of the value itself, therefore simply returning another value of the same type. However, this is not strictly followed, leading to naming inconsistencies such as asUppercase vs. capitalized.

Type conversions asByteArray, asByteString, asDate, asDateAndTime, asDuration, asInteger, asOctetString, asSignedInteger, asString, asStringOrText, asSymbol, asTime, asUnsignedInteger, asWideString

Value transformation or escapement asCamelCase, asComment, asFourCode, asHTMLString, asHex, asLegalSelector, asLowercase, asPluralBasedOn:, asUncommentedCode, asUppercase

Past participles read more fluidly, but they do not always make sense, e.g. commented suggests adding a comment to the receiver, instead of converting it to one. Conversely, adopting as*Something* naming in all cases would be at the price of some contorted English (asCapitalized instead of capitalized).

4. Inconsistencies and Smells

Here we report on the strange things we found and that could be fixed or improved in the short term.

Redundant specializations. Some methods express a very similar intent, but with slightly differing parameters, constraints, or results. When possible, user code should be rewritten in terms of a more general approach; for example, many of the pattern-finding methods could be expressed as regular expression matching.

Substring lookup findAnySubStr:startingAt: and findDelimiters:-startingAt: are synonymous if their first argument is a collection of single-character delimiters; the difference is that the former also accepts string delimiters.

Character lookup indexOfFirstUppercaseCharacter is redundant with SequenceableCollection»findFirst: with very little performance benefit.

Ad-hoc behavior. Ad-hoc methods simply provide convenience behavior that is both specific and little used. Often, the *redundant specialization* also applies.

Numeric suffix numericSuffix has only one sender in the base Pharo image; conversely, it is the only user of stemAndNumericSuffix and endsWithDigit; similarly, endsWithAColon has only one sender.

Finding text findLastOccurrenceOfString:startingAt: has only one sender, related to code loading; findWordStart:startingAt: has no senders.

Find tokens findTokens:escapedBy: has no senders besides tests; findTokens:includes: has only one sender, related to email address detection; findTokens:keep: only has two senders.

Replace tokens copyReplaceTokens:with: has no senders and is convenience for copyReplaceAll:with:asTokens:; redundant with regular expression replacement.

Miscellaneous lineCorrespondingToIndex

Mispackaged or misclassified methods. There are a couple methods that do not really belong to String:

- asHex concatenates the literal notation for each character (*e.g.,* 16r6F) without any separation, producing an ambiguous result; it could be redefined using flatCollect:.
- indexOfSubCollection: should be defined in SequenceableCollection; also, it is eventually implemented in terms of findString:, which handles case, so it is not a simple subsequence lookup.

Many ad-hoc or dubious-looking methods with few senders seem to come from the completion engine; the multiple versions and forks of this package have a history of maintenance problems, and it seems that methods that should have been extensions have been included in the core packages.

Misleading names. Some conversion-like methods are actually encoding or escaping methods: they return another string whose contents match the receiver's, albeit in a different representation (uppercase, lowercase, escaped for comments, as HTML. . .).

Duplicated code. Substring testing methods beginsWithEmpty:caseSensitive: and occursInWithEmpty:caseSensitive: are clearly duplicated: they only differ by a comparison operator. They are also redundant with the generic beginsWith:, except for case-sensitivity. Moreover, the –WithEmpty: part of their selector is confusing; it suggests that argument is supposed to be empty, which makes no sense. Finally, their uses hint that were probably defined for the completion engine and should be packaged there.

5. The ANSI Smalltalk Standard

The ANSI standard defines some elements of the Smalltalk language [14]. It gives the definition *"String literals define objects that represent sequences of characters."* However, there are few guidelines helpful with designing a string API.

The ANSI standard defines the readableString protocol as conforming to the magnitude protocol (which supports the comparison of entities) and to the sequencedReadableCollection protocol, as shown in Figure 2 [14, section 5.7.10]. We present briefly the protocol sequencedReadableCollection.

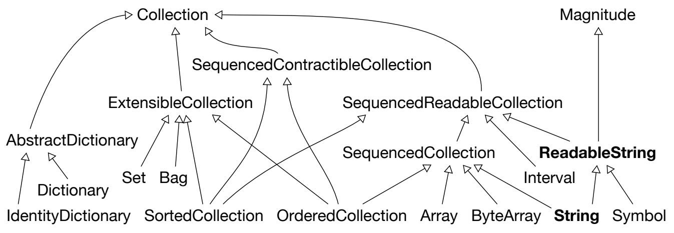

Figure 2: Inheritance of the ANSI Smalltalk protocols.

SequencedReadableCollection. The sequencedReadableCollection protocol conforms to the collection protocol; it provides behavior for reading an ordered collection of objects whose elements can be accessed using external integer keys between one and the number of elements in the collection. It specifies that the compiler should support the following messages — we add some of the argument names for clarity:

Concatenation: , tail (the *comma* binary message)

Equality: = other

Element access: at: index, at: index ifAbsent: block, first, last, before: element, after:, findFirst: block, findLast:

Subsequence access: from: startIndex to: stopIndex do: block

Transforming: reverse

Substitution: copyReplaceAll: elements with: replacingElements, copyReplaceFrom: startIndex to: stopIndex with: replacingElements, copyReplacing: targetElement withObject: replacingElement, copyReplaceFrom: startIndex to: stopIndex withObject: replacingElement

Index of element(s): indexOf: element, indexOf:ifAbsent:, indexOfSubCollection:startingAt:, indexOfSubCollection:startingAt:ifAbsent:

Copy: copyFrom: startIndex to: lastIndex, copyWith: element, copyWithout:

Iteration: do:, from:to:keysAndValuesDo:, keysAndValuesDo:, reverseDo:, with:do:

Many operations require explicit indices that have to be obtained first, making the API not very fluid in practice. Moreover, the naming is often obscure: for example, copyWith: copies the receiver, and *appends* its argument to it.

ReadableString. This protocol provides messages for string operations such as copying, comparing, replacing, converting, indexing, and matching. All objects that conform to the readableString protocol are comparable. The copying messages inherited from the sequencedReadableCollection protocol keep the same behavior. Here is the list of messages:

Concatenation: , (comma)

Comparing: <, <=, >, >=

Converting: asLowercase, asString, asSymbol, asUppercase

Substituing: copyReplaceAll:with:, copyReplaceFrom:to:with:, copyReplacing:withObject:, copyWith:

Subsequence access: subStrings: separatorCharacters

Testing: sameAs:

Analysis and ANSI Compliance. Indices are omnipresent, and very few names are specific to strings as opposed to collections, which makes the protocol feel shallow, low-level and implementation revealing. In particular, because the underlying design is stateful, the copyReplace* messages have to explicitly reveal that they do not modify their receiver through cumbersome names. In a better design, naming would encourage using safe operations over unsafe ones.

We believe that the value added by complying with the ANSI standard is shallow. Indeed, the standard has not been updated to account for evolutions such as immutability, and it does not help building a fluent, modern library. ANSI should not be followed for the design of a modern String library.

6. An Overview of Expected String Features

Different languages do not provide the exact same feature set[3], or the same level of convenience or generality. However, comparing various programming languages, we can identify the main behavioral aspects of strings. Note that these aspects overlap: for instance, transposing a string to upper-case involves substitution, and can be performed in place or return a new string; splitting requires locating separators and extracting parts as smaller strings, and is a form of parsing.

Extracting. Locating or extracting parts of a string can be supported by specifying either explicit indices, or by matching contents with various levels of expressiveness: ad-hoc pattern, character ranges, regular expressions.

Splitting. Splitting strings into chunks is the basis of simple parsing and string manipulation techniques, like counting words or lines in text. To be useful, splitting often needs to account for representation idiosyncrasies like which characters count as word separators or the different carriage return conventions.

Merging. The reverse of splitting is merging several strings into one, either by concatenation of two strings, or by joining a collection of strings one after another, possibly with separators.

Substituting. The popularity of Perl was built on its powerful pattern-matching and substitution features. The difficulty with substitution is how the API conveys whether one, many, or all occurrences are replaced, and whether a sequence of elements or a single element is replaced.

[3]They can even rely on specific syntax, like Ruby's string interpolation.

Testing. Strings provide many predicates, most importantly determining emptiness, or inclusion of a particular substring, prefix or suffix. Other predicates range from representation concerns, like determining if all characters belong to the ASCII subset, or of a more ad-hoc nature, like checking if the string is all uppercase or parses as an identifier.

Iterating. Strings are often treated as collections of items. In Pharo a string is a collection of characters and as such it inherits all the high-level iterators defined in SequenceableCollection and subclasses. Similarly, Haskell's Data.String is quite terse (just 4 or so functions), but since strings are Lists, the whole panoply of higher-level functions on lists are available: foldr, map, etc.

Endogenous conversion. Strings can be transformed into other strings according to domain-specific rules: this covers encoding and escaping, case transpositions, pretty-printing, natural language inflexion, etc.

Exogenous conversion. Since strings serve as a human-readable representation or serialization format, they can be parsed back into non-string types such as numbers, URLs, or file paths.

Mutating vs copying. Strings may be considered as collections and provide methods to modify their contents in-place, as opposed to returning a new string with different contents from the original. Note that this point is orthogonal to the other ones, but influences the design of the whole library.

Mutating strings is dangerous, because strings are often used as value objects, and it is not clear at first sight if a method has side-effects or not. For example, in translateToUppercase, the imperative form hints that it is an in-place modification, but not in trim. Also, safe transformations often rely on their side-effect counterpart: for instance, the safe asUppercase sends translateToUppercase to a copy of its receiver.

In the case of strings, we believe methods with side effects should be clearly labeled as low-level or private, and their use discouraged; moreover, a clear and systematic naming convention indicating the mutable behavior of a method would be a real plus. Finally, future developments of the Pharo VM include the Spur object format, which supports immutable instances; this is an opportunity to make literal strings safe[4], and to reduce copying by sharing character data between strings.

7. Strings in Other Languages

To support the analysis and redesign of the current string libraries in Pharo, we analysed the situation in several other languages. We took two criteria into account to select the languages below: mainstream object-oriented languages but also new languages showing alternative designs. Indeed, our study is about the design of the API at the level of features, how they compose together and in relation with other types, and how they are organized in terms of individual methods or functions. In

[4]While clever uses for mutable literals have been demonstrated in the past, we think it is a surprising feature and should not be enabled by default.

that light, we believe that the underlying programming paradigm is just one of many factors that influence the API design. For instance, it would be possible and probably desirable to have fewer side-effects and more declarative method names in Pharo's string API, resulting in a style much closer to functional programming than the current string implementation; Haskell, with its own limits, provides a worthwile reference point in that direction.

We will present the key characteristics of the design of strings in Haskell, Java, Python, Ruby, and Rust. Then we will discuss some of the used design.

7.1. Haskell

In Haskell, the default string implementation Data.String is actually a linked list of characters. This was a design choice to reuse the existing pattern matching and list manipulation functions with virtually no string-specific code; but it is also known to have a huge space overhead and bad performance characteristics for usual string use. However, if we look further than the core libraries that come with GHC, the Haskell Platform distribution also provides Data.Text, an implementation of strings based on a packed array of UTF-16 codepoints. The same package also includes a lazy variant of that data structure.

In terms of interfaces, Data.List[5] and Data.Text[6] are of similar sizes (respectively 116 and 94 functions), but share 60 functions in common, including Data.Text.append and Data.Text.index which are defined as the (++) and (!!)) operators in Data.List (see Table 1). This is because many list functions do not apply to lists of characters: lookup expects an association list, and & or expect lists of booleans, sum expects a list of numbers, etc. Conversely, Data.Text defines additional functions that are related to formatting (center, justifyLeft, toLower, toTitle), cleaning up (dropAround, strip), or parsing text (breakOn, split), or parsing text (breakOn, split).

7.2. Java

In Java, instances of the String class are immutable (See Table 2). This means that strings can be shared, but also that concatenating them allocates and copies memory. To build complex strings while limiting memory churn, the standard library provides StringBuilder and StringBuffer; both have the exact same interface, except the latter is thread-safe. Finally, CharSequence is an interface which groups a few methods for simple read-only access to string-like objects; it seems like it has a similar purpose as Rust's slices, but Java strings do not appear to share their underlying character data: subSequence() is the same as substring(), which copies the required range of characters.

Third-party libraries such as Apache Commons[7] provide additional string-related methods in utility classes such as StringUtils. However, since those classes only define static methods, they do not lend themselves to late binding and polymorphism.

Haskell— 60 functions common to both Data.List and Data.Text:

(!!) index	findIndex	intercalate	minimum	tail
(++) append	foldl	intersperse	null	tails
all	foldl'	isInfixOf	partition	take
any	foldl1	isPrefixOf	replicate	takeWhile
break	foldl1'	isSuffixOf	reverse	transpose
concat	foldr	last	scanl	uncons
concatMap	foldr1	length	scanl1	unfoldr
drop	group	lines	scanr	unlines
dropWhile	groupBy	map	scanr1	unwords
dropWhileEnd	head	mapAccumL	span	words
filter	init	mapAccumR	splitAt	zip
find	inits	maximum	stripPrefix	zipWith

56 functions specific to Data.List:

(\\)	genericSplitAt	or	unzip4
and	genericTake	permutations	unzip5
cycle	insert	product	unzip6
delete	insertBy	repeat	unzip7
deleteBy	intersect	scanl'	zip3
deleteFirstsBy	intersectBy	sort	zip4
elem	isSubsequenceOf	sortBy	zip5
elemIndex	iterate	sortOn	zip6
elemIndices	lookup	subsequences	zip7
findIndices	maximumBy	sum	zipWith3
genericDrop	minimumBy	union	zipWith4
genericIndex	notElem	unionBy	zipWith5
genericLength	nub	unzip	zipWith6
genericReplicate	nubBy	unzip3	zipWith7

34 functions specific to Data.Text:

breakOn	count	snoc	toCaseFold
breakOnAll	dropAround	split	toLower
breakOnEnd	dropEnd	splitOn	toTitle
center	empty	strip	toUpper
chunksOf	justifyLeft	stripEnd	unfoldrN
commonPrefixes	justifyRight	stripStart	unpack
compareLength	pack	stripSuffix	unpackCString#
cons	replace	takeEnd	
copy	singleton	takeWhileEnd	

Table 1: Functions defined by Haskell modules Data.List and Data.Text

7.3. Python

Python's string type is str[8], an immutable sequence of Unicode codepoints, whose methods are listed in Table 3. Besides those methods, it also inherits special methods that implement the behavior for the sequence-related expressions (index-based access, count and presence of elements). A few additional functions are defined in module string[9], most notably printf-style formatting, and Python also provides io.StringIO, a stream-like object to compose large strings efficiently, but this provides a limited API similar to a file stream, unlike Java's StringBuilder which supports insertion and replace operations.

The general impression is that the API is pretty terse, especially since there are some symmetric sets of methods, *i.e.,*strip, lstrip, rstrip. Some methods seem too specialized to be present in such a small API (*e.g.,*swapcase, title, istitle).

Finally, since Python, like Ruby, does not have an individual character type, some character-specific behavior is reported on strings: out of 11 predicates, only two really apply specifically

[5]https://hackage.haskell.org/package/base-4.9.0.0/docs/Data-List.html
[6]https://hackage.haskell.org/package/text-1.2.2.1/docs/Data-Text.html
[7]https://commons.apache.org

[8]https://docs.python.org/3.6/library/stdtypes.html#textseq
[9]https://docs.python.org/3.6/library/string.html

Java— 35 methods in String:

charAt	endsWith	lastIndexOf	startsWith
codePointAt	equals	length	subSequence
codePointBefore	equalsIgnoreCase	matches	substring
codePointCount	getBytes	offsetByCodePoints	toCharArray
compareTo	getChars	regionMatches	toLowerCase
compareToIgnoreCase		replace	toString
concat	indexOf	replaceAll	toUpperCase
contains	intern	replaceFirst	trim
contentEquals	isEmpty	split	hashCode

24 methods in StringBuffer/StringBuilder:

append	codePointCount	insert	setCharAt
appendCodePoint	delete	lastIndexOf	setLength
capacity	deleteCharAt	length	subSequence
charAt	ensureCapacity	offsetByCodePoints	substring
codePointAt	getChars	replace	toString
codePointBefore	indexOf	reverse	trimToSize

Table 2: Methods defined in Java on string-like classes

Python— 42 methods in str:

capitalize	find	isdigit	isupper	rfind	startswith
casefold	format	isidentifier	join	rindex	strip
center	format_map	islower	ljust	rjust	swapcase
count	index	isnumeric	lower	rpartition	title
encode	isalnum	isprintable	lstrip	rstrip	translate
endswith	isalpha	isspace	partition	split	upper
expandtabs	isdecimal	istitle	replace	splitlines	zfill

Table 3: Methods defined in Python on the str text sequence type

to strings (isidentifier and istitle), the other 9 being universally quantified character predicates. Encoding and decoding between bytes and Unicode strings is done via the str.encode() and bytes.decode() methods, which rely on another package: codecs; here again, character-specific or encoding-specific behavior does not seem to exist as first-class objects, as codecs are specified by name (strings).

7.4. Ruby

Ruby's strings are mutable sequence of bytes[10]; however, each String instance knows its own encoding. Ruby's message send syntax is quite expressive, and many of its APIs make extensive use of optional parameters and runtime type cases to provide behavior variants.

A first example is the convention that iteration methods each_byte, each_char, each_codepoint, and each_line either behave as an internal iterator (*i.e.,* a higher-order function) when passed a block, or return an enumerator object when the block is omitted (external iteration).

A second example is the [] method, which implements the square bracket notation for array access; on strings, this is used for substring extraction, and accepts a number of parameter patterns:

- a single index, returning a substring of length one (Ruby does not have an individual character type),
- a start index and an explicit length,

[10]http://www.rubydoc.info/stdlib/core/String

Ruby— 116 methods in String:

%	codepoints	initialize	size
*	concat	replace	slice (!)
+	count	insert	split
−	crypt	inspect	squeeze (!)
<<	delete (!)	intern	start_with?
<=>	downcase (!)	length	strip (!)
==	dump	lines	sub (!)
===	each_byte	ljust	succ (!)
=~	each_char	lstrip (!)	sum
[]	each_codepoint	match	swapcase (!)
[]=	each_line	next (!)	to_c
ascii_only?	empty?	oct	to_f
b	encode (!)	ord	to_i
bytes	encoding	partition	to_r
bytesize	end_with?	prepend	to_s
byteslice	eql?	replace	to_str
capitalize (!)	force_encoding	reverse (!)	to_sym
casecmp	freeze	rindex	tr (!)
center	getbyte	rjust	tr_s (!)
chars	gsub (!)	rpartition	unpack
chomp (!)	hash	rstrip (!)	upcase (!)
chop (!)	hex	scan	upto
chr	include?	scrub (!)	valid_encoding?
clear	index	setbyte	

Table 4: Methods defined in Ruby's String class. Methods marked with (!) have an associated in-place version following the Ruby naming convention; e.g. upcase returns an uppercased copy while upcase! modifies the receiver in-place.

- a range object, locating the substring by start/end bounds instead of by its length,
- a regular expression, optionally with a capture group specifying which part of the matched substring to return,
- another string, returning it if it occurs in the receiver.

Note also that indices can be negative, in which case they are relative to the end of the string.

Another widely adopted naming convention in Ruby is that methods with names terminated by an exclamation point modify their receiver in-place instead of returning a modified copy; strings are a nice example of this pattern, as more than a third of the methods belong to such copy/in-place pairs.

7.5. Rust

Rust has two main types for character strings: *string slices*, represented by the pointer type &str[11], and the boxed type String[12] (Table 5). Both types store their contents as UTF-8 bytes; however, while String is an independent object that owns its data, allocates it on the heap and grows it as needed, &str is a view over a range of UTF-8 data that it does not own itself. Literal strings in Rust code are immutable &str slices over statically-allocated character data.

Making a String from a &str slice thus requires allocating a new object and copying the character data, while the reverse operation is cheap. In fact, the compiler will implicitly cast a String into a &str as needed, which means that in practice,

[11]https://doc.rust-lang.org/std/primitive.str.html
[12]https://doc.rust-lang.org/std/string/struct.String.html

Rust— 43 methods defined on string slices &str:			
as_bytes	find	rfind	splitn
as_ptr	into_string	rmatch_indices	starts_with
bytes	is_char_boundary	rmatches	to_lowercase
char_indices	is_empty	rsplit	to_uppercase
chars	len	rsplit_terminator	trim
contains	lines	rsplitn	trim_left
encode_utf16	match_indices	split	trim_left_matches
ends_with	matches	split_at	trim_matches
escape_debug	parse	split_at_mut	trim_right
escape_default	replace	split_terminator	trim_right_matches
escape_unicode	replacen	split_whitespace	
26 methods defined on the boxed String type:			
as_bytes	from_utf16_lossy	is_empty	reserve
as_mut_str	from_utf8	len	reserve_exact
as_str	from_utf8_lossy	new	shrink_to_fit
capacity	insert	pop	truncate
clear	insert_str	push	with_capacity
drain	into_boxed_str	push_str	
from_utf16	into_bytes	remove	

Table 5: Methods defined in Rust on strings and string slices

all methods of slices are also available on boxed strings, and String only adds methods that are concerned with the concrete implementation.

An surprising design decision in Rust is that strings do *not* implement the array-like indexing operator. Instead, to access the contents of a string, the library requires explicit use of iterators. This motivated by the tension between the need, as a systems programming language, to have precise control of memory operations, and the fact that practical, modern encodings (be it UTF-8 or UTF-16) encode characters into a varying number of bytes. Variable-length encoding makes indexed access to individual characters via dead-reckoning impossible: since the byte index depends on the space occupied by all preceding characters, one has to iterate from the start of the string. The implications are two-fold: first, this design upholds the convention that array-like indexing is a constant-time operation returning values of fixed size. Second, multiple iterators are provided on equal footing (methods bytes(), chars(), lines(), or split()), each of them revealing a different abstraction level, with no intrinsic or default meaning for what the *n*-th element of a string is; this also makes the interface more uniform.

8. Reflection on String APIs

It is difficult to form an opinion on the design of an API before getting feedback from at least one implementation attempt. Still, at this stage, we can raise some high level points that future implementors may consider. We start by discussing some issues raised in the analysis of the previous languages, then we sketch some proposals for a future implementation.

8.1. Various APIs in Perspective

While proper assessment of API designs would be more suited for a publication in cognitive sciences, putting a few languages in perspective during this cursory examination of the string API raised a few questions.

Python:	
ord('a')	⇒ 97
ord('abc')	TypeError: ord() expected a character, but string of length 3 found
ord('')	TypeError: ord() expected a character, but string of length 0 found
Ruby:	
?a.class	⇒ String
?a.ord	⇒ 97
'a'.ord	⇒ 97
'abc'.ord	⇒ 97
''.ord	ArgumentError: empty string

Table 6: Character / string confusion in Python and Ruby. Both languages use degenerate strings in place of characters; Ruby does have a literal character syntax, but it still represents a one-character string.

First-class characters or codepoints. In Ruby or Python, characters are strings of length one, which has strange implications on some methods, as shown in table 6. Was that choice made because the concept of character or codepoint was deemed useless? If so, is it due to lack of need in concrete use-cases, or due to early technical simplifications, technical debt and lack of incentives to change? If not, is it undertaken by separate encoding-related code, or by strings, even though it will be often used on degenerate single-character instances? There is a consensus nowadays around Unicode, which makes encoding conversions a less pressing issue; however, Unicode comes with enough complexities of its own —without even considering typography— that it seems a dedicated character/codepoint type would be useful. For instance, Javascript implements strings as arrays of 16-bit integers to be interpreted as UTF-16, but without taking surrogate sequences into account, which means that the length method is not guaranteed to always return the actual number of characters in a string.

Sharing character data. Second, there is a compromise between expressivity and control over side effects, data copying, and memory allocation. Many applications with heavy reliance on strings (*e.g.,* parsers, web servers) benefit from sharing character data across several string instances, because of gains both in memory space and in throughput; however, this requires that the shared data does not change. In this regard, Rust's string slices are interesting because they provide substrings of constant size and creation time without adding complexity to the API. Conversely, Haskell's lazy string compositions, or data structures like ropes, provide the equivalent for concatenation, without a distinct interface like Java's StringBuilder.

Matching and regular patterns. Regular patterns, in languages where they are readily available, are highly effective at analyzing strings. We did not discuss them here, because while they are a sister feature of strings, they really are a domain-specific language for working on strings that can be modularized independently, much like full-fledged parsers. A way to do that is to make regular expressions polymorphic with other string-accessing types such as indices, ranges, or strings as patterns); Ruby does this by accepting various types as argument of its indexing/substring methods, and Rust by defining a proper abstract type Pattern that regular patterns implement.

8.2. Concerns for a New String Implementation

For an API to provide rich behavior without incurring too much cognitive load, it has to be regular and composable.

Strings and characters are different concepts. The distinction between character and string types distributes functionality in adequate abstractions. Characters or codepoints can offer behavior related to their encoding, or even typographic or linguistic information such as which alphabet they belong to.

Note that the implementation does not have to be naive and use full-fledged character objets everywhere. In Pharo, String is implemented as a byte or word array in a low-level encoding, and Character instances are only created on demand. Most importantly, a character is not a mere limited-range integer. In this regard, the design of Rust validates that design choice.

Strings are sequences, but not collections. Strings differ from usual lists or arrays in that containing a specific element does not really matter *per se*; instead, their contents have to be interpreted or parsed. We think this is why their iteration interface is both rich and ad hoc, and follows many arbitrary contextual conventions like character classes or capitalization. From this perspective, we should probably reconsider the historical design choice to have String be a Collection subclass.

Iterations. Strings represent complex data which can be queried, navigated, iterated in multiple ways (bytes, characters, words, lines, regular expression matches…).

Iteration based on higher-order functions is an obvious step in this direction; Smalltalk dialects use internal iterators as the iconic style to express and compose iterations, but this seems to have discouraged the appearance of an expressive set of streaming or lazy abstractions like Ruby's enumerators or Rust's iterators. Therefore, external iterators should be investigated, under the assumption that extracting the control flow may lead to better composeability. Of course, co-design between strings and collection/stream libraries would be beneficial.

Encodings. It is misguided to assume that characters always directly map to bytes, or that any sequence of bytes can be viewed as characters. To bridge bytes and characters, encodings are required; the API should take them into account explicitly, including provisions for impossible conversions and probably for iteration of string contents simultaneously as characters and as encoded data.

String buffers and value strings. Pharo strings currently have a single mutable implementation which is used in two distinct roles: as a value for querying and composing, and as a buffer for in-place operations. Streams can assemble large strings efficiently, but more complex editing operations rely on fast data copying because of the underlying array representation.

Distinguishing these two roles would allow for internal representations more suited to each job and for a more focused API. In particular, the guarantees offered by immutable strings and views like Rust's slices open many possibilities for reducing data copies and temporary object allocations.

Consistency and cleanups. Finally, we would like to consolidate close methods into consistently named groups or even chains of methods whenever possible. Immutable strings would favor a declarative naming style.

The current implementation suffers from the presence of many ad-hoc convenience methods, many of which do not belong in the core API of strings and should be extracted or removed.

Several methods are related to converting between strings and other kinds of objects or values. These conversion methods come in a limited set that is neither generic nor complete; instead we would prefer a clear, generic, but moldable API for parsing instances of arbitrary classes out of their string representations.

9. Discussion and Perspectives

In this paper, we assess the design of character strings in Pharo. While strings are simple data structures, their interface is surprisingly large. Indeed, strings are not simple collections of elements; they can be seen both as explicit sequences of characters, and as simple but very expressive values from the domain of a language or syntax. In both cases, strings have to provide a spectrum of operations with many intertwined characteristics: abstraction or specialization, flexibility or convenience. We analyze the domain and the current implementation to identify recurring idioms and smells.

The idioms and smells we list here deal with code readability and reuseability at the level of messages and methods; they fall in the same scope as Kent Beck's list [5]. While the paper focuses on strings, the idioms we identify are not specific to strings, but to collections, iteration, or parameter passing; modulo differences in syntax and style usages, they apply to other libraries or object-oriented programming languages. To identify the idioms and smells, we rely mostly on code reading and the usual tools provided by the Smalltalk environment. This is necessary in the discovery stage, but it raises several questions:

- How to document groups of methods that participate in a given idiom? As we say in Section 2, method protocols are not suitable: they partition methods by feature or theme, but idioms are overlapping patterns of code factorization and object interaction.

- How to specify, detect, check, and enforce idioms in the code? This is related to architecture conformance techniques [15].

[1] J. Blanchette, The little manual of API design, http://www4.in.tum.de/~blanchet/api-design.pdf (Jun. 2008).

[2] J. Stylos, S. Clarke, B. Myers, Comparing API design choices with usability studies: A case study and future directions, in: P. Romero, J. Good, E. A. Chaparro, S. Bryant (Eds.), 18th Workshop of the Psychology of Programming Interest Group, University of Sussex, 2006, pp. 131–139. doi:10.1.1.102.8525.

[3] J. Stylos, B. Myers, Mapping the space of API design decisions, in: IEEE Symposium on Visual Languages and Human-Centric Computing, 2007, pp. 50–57. doi:10.1109/VLHCC.2007.44.

[4] M. Piccioni, C. A. Furia, B. Meyer, An empirical study of API usability, in: IEEE/ACM Symposium on Empirical Software Engineering and Measurement, 2013. doi:10.1109/ESEM.2013.14.

[5] K. Beck, Smalltalk Best Practice Patterns, Prentice-Hall, 1997. URL http://stephane.ducasse.free.fr/FreeBooks/BestSmalltalkPractices/Draft-Smalltalk%20Best%20Practice%20Patterns%20Kent%20Beck.pdf

[6] K. Cwalina, B. Abrams, Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .Net Libraries, 1st Edition, Addison-Wesley Professional, 2005.

[7] J. Bloch, How to design a good api and why it matters, in: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications, OOPSLA '06, ACM, 2006, pp. 506–507. doi:10.1145/1176617.1176622. URL http://doi.acm.org/10.1145/1176617.1176622

[8] R. E. Griswold, M. T. Griswold, The Icon Programming Language, Peer-to-Peer Communications, 1996. URL http://www.peer-to-peer.com/catalog/language/icon.html

[9] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Classboxes: Controlling visibility of class extensions, Journal of Computer Languages, Systems and Structures 31 (3-4) (2005) 107–126. doi:10.1016/j.cl.2004.11.002. URL http://rmod.inria.fr/archives/papers/Berg05a-CompLangESUG04-classboxesJournal.pdf

[10] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000, pp. 130–145.

[11] B. Woolf, Null object, in: R. Martin, D. Riehle, F. Buschmann (Eds.), Pattern Languages of Program Design 3, Addison Wesley, 1998, pp. 5–18.

[12] S. Murer, S. Omohundro, D. Stoutamire, C. Szyperski, Iteration abstraction in sather, ACM Transactions on Programming Languages and Systems 18 (1) (1996) 1–15. doi:10.1145/225540.225541.

[13] R. Hickey, Clojure transducers, http://clojure.org/transducers.

[14] ANSI, New York, American National Standard for Information Systems – Programming Languages – Smalltalk, ANSI/INCITS 319-1998, http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf (1998).

[15] S. Ducasse, D. Pollet, Software architecture reconstruction: A process-oriented taxonomy, IEEE Transactions on Software Engineering 35 (4) (2009) 573–591. doi:10.1109/TSE.2009.19. URL http://rmod.inria.fr/archives/papers/Duca09c-TSE-SOAArchitectureExtraction.pdf

Appendix — Classifying the Pharo String API

Finding

Methods returning places in the string (indices, ranges).

findString:	findString:startingAt:
findString:startingAt:caseSensitive:	
findLastOccurrenceOfString:startingAt:	
allRangesOfSubString:	findAnySubStr:startingAt:
findCloseParenthesisFor:	findDelimiters:startingAt:
findWordStart:startingAt:	no senders
findIn:startingAt:matchTable:	auxiliary method
findSubstring:in:startingAt:matchTable:	auxiliary method
findSubstringViaPrimitive:in:startingAt:matchTable:	one sender

indexOf: indexOf:startingAt: indexOf:startingAt:ifAbsent:

indexOfSubCollection:	mispackaged
indexOfSubCollection:startingAt:ifAbsent:	
indexOfFirstUppercaseCharacter	redundant, one sender
indexOfWideCharacterFrom:to:	
lastSpacePosition	
lastIndexOfPKSignature:	adhoc or mispackaged
skipAnySubStr:startingAt:	skipDelimiters:startingAt:

Extracting

Methods returning particular substrings.

wordBefore:	
findSelector	mispackaged, specific to code browser
findTokens:	
findTokens:escapedBy:	no senders (besides tests)
findTokens:includes:	one sender
findTokens:keep:	
lineCorrespondingToIndex:	
squeezeOutNumber	ugly parser, one sender
splitInteger	what is the use-case?
stemAndNumericSuffix	duplicates previous method

Splitting

Methods returning a collection of substrings.

lines	
subStrings:	
substrings	not a call to previous one, why?
findBetweenSubStrs:	
keywords	adhoc, assumes receiver is a selector

Enumerating

linesDo: lineIndicesDo: tabDelimitedFieldsDo:

Conversion to other objects

Many core classes such as time, date and duration that have a compact and meaningful textual description extend the class String to offer conversion from a string to their objects. Most of them could be packaged with the classes they refer to, but splitting a tiny core into even smaller pieces does not make a lot of sense, and there are legitimate circular dependencies in the core: a string implementation cannot work without integers, for example. Therefore, most of these methods are part of the string API from the core language point of view:

asDate	asNumber	asString	asSymbol
asTime	asInteger	asStringOrText	
asDuration	asSignedInteger	asByteArray	
asDateAndTime	asTimeStamp		

Some other methods are not as essential:

asFourCode romanNumber string stringhash

Conversion between strings

A different set of conversion operations occurs between strings themselves.

- typography and natural language: asLowercase, asUppercase, capitalized, asCamelCase, withFirstCharacterDownshifted, asPluralBasedOn:, translated, translatedIfCorresponds, translatedTo:
- content formatting: asHTMLString, asHex, asSmalltalkComment, asUncommentedSmalltalkCode,
- internal representation: asByteString, asWideString, asOctetString

Streaming

printOn:	putOn:	storeOn:

Comparing

caseInsensitiveLessOrEqual:	caseSensitiveLessOrEqual:
compare:with:collated:	compare:caseSensitive:
compare:	sameAs:

Testing

endsWith:	endsWithAnyOf:		
startsWithDigit	endsWithDigit	endsWithAColon	ad-hoc
hasContentsInExplorer			should be an extension
includesSubstring:caseSensitive:		includesSubstring:	
includesUnifiedCharacter		hasWideCharacterFrom:to:	
isAllDigits	isAllSeparators	isAllAlphaNumerics	
onlyLetters			inconsistent name
isString	isAsciiString	isLiteral	
isByteString	isOctetString	isLiteralSymbol	
isWideString			
beginsWithEmpty:caseSensitive:			bad name, duplicate
occursInWithEmpty:caseSensitive:			bad name, mispackaged

Querying

lineCount	lineNumber:		
lineNumberCorrespondingToIndex:		leadingCharRunLengthAt:	
initialIntegerOrNil	numericSuffix	indentationIfBlank:	
numArgs			selector-related
parseLiterals			contents of a literal array syntax

Substituting

copyReplaceAll:with:asTokens:	copyReplaceTokens:with:	
expandMacros	expandMacrosWithArguments:	
expandMacrosWith:	expandMacrosWith:with:	
expandMacrosWith:with:with:		
expandMacrosWith:with:with:with:		
format:		
replaceFrom:to:with:startingAt:		primitive
translateWith:	translateFrom:to:table:	
translateToLowercase	translateToUppercase	

Correcting

correctAgainst:	correctAgainst:continuedFrom:
correctAgainstDictionary:continuedFrom:	
correctAgainstEnumerator:continuedFrom:	

Operations

contractTo:	truncateTo:	
truncateWithElipsisTo:		
encompassLine:	encompassParagraph:	
withNoLineLongerThan:		
withSeparatorsCompacted	withBlanksCondensed	
withoutQuoting		
withoutLeadingDigits	withoutTrailingDigits	
withoutPeriodSuffix	withoutTrailingNewlines	
padLeftTo:	padLeftTo:with:	
padRightTo:	padRightTo:with:	
padded:to:with:		duplicates the two previous
surroundedBy:	surroundedBySingleQuotes	
trimLeft:right:	trim	trimmed
trimLeft	trimBoth	trimRight
trimLeft:	trimBoth:	trimRight:

Encoding

convertFromEncoding:	convertFromWithConverter:
convertToEncoding:	convertToWithConverter:
convertToSystemString	encodeDoublingQuoteOn:
withLineEndings:	withSqueakLineEndings
withUnixLineEndings	withInternetLineEndings
withCRs	convenience, used a lot

Matching

alike:	howManyMatch:	similarity metrics
charactersExactlyMatching:		bad name: common prefix length
match:	startingAt:match:startingAt:	

Low-Level Internals

hash	typeTable	
byteSize	byteAt:	byteAt:put:
writeLeadingCharRunsOn:		

Candidates for removal

While performing this analysis we identified some possibly obsolete methods.

asPathName	asIdentifier:	asLegalSelector
do:toFieldNumber:		
indexOfFirstUppercaseCharacter		