Improving API Documentation for Java-like Languages

Gilles Dubochet Donna Malayeri

Ecole Polytechnique Fédérale de Lausanne
Lausanne, Switzerland
{firstname.lastname}@epfl.ch

Abstract

The Javadoc paradigm for displaying API documentation to users is quite popular, with similar variants existing for many mainstream languages. However, two user interface design properties of Javadoc may reduce its utility when displaying documentation for APIs that make use of inheritance and parametric polymorphism. First, Javadoc does not show a flattened view of all members of a class or interface, but rather only those defined directly in the type. Second, and as a consequence, any methods whose types contain type parameters of a superclass will always be shown in the context of the superclass. That is, if the method $C.m$ returns type `T`, subclasses of C will always see this parent signature, even if they instantiate `T` to a concrete type such as `Integer`.

We show that this situation arises often in some libraries, and present the results of a study that measures the usability consequences of these two Javadoc design decisions. Our results show that a user interface that shows instantiated type parameters for members is preferred over one that presents type parameters in the Javadoc style.

1. Introduction

Javadoc-style documentation is a popular method for presenting API documentation to users with similar tools in existence for most mainstream languages. However, producing documentation for a language that makes heavy use of inheritance and parametric polymorphism can be challenging.

In particular, Javadoc lists only those members defined (or overridden) directly in a class or interface. Inherited members are listed only with their name (not even a full signature) and link to the documentation in the parent class or interface. This presents the problem that it can be difficult to find a particular method if one does not know in which supertype it was defined.

Additionally, with increased use of parametric polymorphism (i.e., generics), documenting method signatures can become quite complex. This is particularly a problem when subclasses instantiate type parameters of their parent class and, as a result, method signatures become more specialized in the subclass. If only a superclass view of the method is shown, the method signature will contain type parameters of the parent class, which can be confusing when seen in the context of the subclass.

1.1 Background

The Scala 2.8 collections library [4, 5] makes heavy use of parametric polymorphism and inheritance. Methods that exist in several classes in the library are typically defined in a common supertype, to maximize code reuse and provide useful abstractions for library clients.

For example, the method `filter`, which returns all of the elements of a collection for which a particular predicate holds, appears in many collection classes. The type of collection that is returned is the same as that of the original collection; applying `filter` to a `List` returns a new `List`, whereas applying it to a `Set` will return a `Set`.

To define `filter` in a common supertype of `List` and `Set` and also retain the most precise possible signature, the library uses a type parameter to represent the collection result type. In particular, in the trait `TraversableLike`, a common supertype of `List` and `Set`, is defined approximately as follows:

```
trait TraversableLike[A, Repr] {
  ...
  def filter[B](p: (A) => Boolean): Repr
}
```

The class `List[A]` inherits (indirectly) from `TraversableLike[A, List[A]]`; `Set` has an analogous declaration.

Unfortunately, this extensive use of inheritance and polymorphism creates a challenge for providing documentation to users. A user who wishes to see the methods that can be called on a `List` does not necessarily care that some methods are implemented in `TraversableLike`. Additionally, the type parameter `Repr` is interesting in the context of

Table 1. Frequency of methods with changed signatures in subclasses in various Scala and Java programs. The first column is the sum of all defined and inherited methods of all declared classes, interfaces, and (for Scala programs) traits.

		Inherited methods		Changed signature		
	Methods	Total	Percentage	Total	% of all methods	% of inherited
Scala standard library	121370	84121	69%	27618	23%	33%
Scala compiler	64310	33983	53%	7368	11%	22%
scalaz[1]	10947	2626	24%	311	3%	12%
Google Data API[2]	96487	67804	70%	16056	17%	24%
java.util	9486	3482	37%	277	3%	8%
plt utilities[3]	12030	5465	45%	284	2%	5%
Google collections[4]	3506	804	23%	44	1%	5%

[1] `http://code.google.com/p/scalaz/`
[2] `http://code.google.com/p/gdata-java-client/`
[3] `https://drjava.svn.sourceforge.net/svnroot/drjava/trunk/plt/`
[4] `http://code.google.com/p/google-collections/`

`TraversableLike`, but not in the context of `List`, as there it has been instantiated to `List[A]`. As this pattern occurs often in the Scala library, browsing Javadoc-style documentation became quite difficult for users, as it involved many click-throughs to parent classes and obscure type parameters [3].

This problem is not confined to the Scala collections library; some Java libraries also use generics in conjunction with inheritance in a similar manner. One such example is the Google Data API,[1] which provides functionality for accessing Google services such as YouTube and Google Calendar. A common pattern in the Java version of this API is to put common functionality in superclasses that take a type parameter to represent the concrete subclass. For example, consider the declaration of the class `BaseEventFeed`:

```
class BaseEventFeed
  <F extends BaseEventFeed<F,E>,
   E extends BaseEventEntry<E>>
```

Concrete subclasses pass themselves as a parameter to `BaseEventFeed`:

```
class CalendarEventFeed extends
  BaseEventFeed
    <CalendarEventFeed,
    CalendarEventEntry>
```

But, since a great deal of functionality is defined in `BaseEventFeed`, users must view those methods in terms of the `F` and `E` type parameters of `BaseEventFeed`, even if they are interested in a concrete subclass such as `CalendarEventFeed`. For example, the method `getEntries` returns `List<E>`. For `CalendarEventFeed`, this will actually be equivalent to `List<CalendarEventFeed>`, but this type is not shown anywhere in the Javadoc.

[1] `http://code.google.com/p/gdata-java-client/`

We ran a simple analysis on several Scala and Java programs to see how often method signatures changed in this manner. Results are displayed in Table 1. The Scala standard library had the highest frequency of methods whose signatures changed in subclasses: 23% of all methods, which corresponds to 33% of inherited methods. Of the Java programs, The Google Data API had the highest frequency of methods with changed signatures, with 17% of all methods (corresponding to 24% of inherited methods) changing signatures in subclasses. This data shows that methods do indeed have different signatures in subclasses and that some programs have a significant number of inherited methods.

The data in Table 1 led us to hypothesize that it would be useful to specialize method types in subclasses, rather than showing types relative to parent type parameters.

Additionally, this data shows that in some programs, a large percentage of methods are inherited from superclasses. But, unless a class is being subclassed, most of its clients do not care whether a method was inherited (or where it was inherited from), but are more interested in the operations that are available on the class. So, we additionally hypothesized that for programs that make substantial use of inheritance, it could be useful to show a flattened list of methods (i.e., all defined and inherited methods) for a given class. Our concrete hypotheses are summarized below.

1.2 Hypotheses

We have the following hypotheses regarding API documentation for Java-like languages:

Hypothesis I. It is more efficient to show a flattened, alphabetical list of methods for each class rather than grouping methods by superclass.

Hypothesis II. It is more efficient to show context-dependent method types (i.e., instantiate type parameters) than to

Version	Description
alpha	Alphabetical list of methods with context-dependent method signatures
inherit	List of methods grouped by parent class, with context-dependent method signatures (i.e., substituted type variables)
javadoc	List of methods grouped by parent class, with superclass-relative method signatures (i.e., unsubstituted type variables)

Table 2. UI modes

show method types in terms of superclass type parameters.

Here, "more efficient" means that users will answer questions more quickly and more correctly using a user interface that has the described property. Additionally, we hypothesize that users will prefer user interfaces with each of above properties.

1.3 Contributions

- Identification of two key design considerations in the design of documentation for Java-like languages (displaying a flattened list of members and instantiated type parameters).
- Results of a study that show that showing instantiated type parameters in a documentation browser leads to increased user satisfaction, when using either a Java or Scala API.

2. Study Design

To measure the effect of each hypothesis independently, there are four possible user interface versions, but only three are sensible, those displayed in Table 2. We do not consider a version consisting of an alphabetical list of methods with superclass-relative method signatures, as this is not semantically meaningful. In particular, if a class C inherits a method from a non-immediate superclass A, the type parameters of A are not obvious within the context of C. Therefore, the only reasonable way to present the type of of any methods inherited from A is to show them in terms of C's, rather than A's, type parameters.

An example of types presented in the first two modes as compared to the last mode is presented in Fig. 1. Note that the types in the "javadoc" mode are presented in terms of parent type parameters.

2.1 Subjects

Subjects were recruited through an announcement on the main Scala mailing list, `scala@listes.epfl.ch`, which advertised a study regarding the usability of Scaladoc. Subjects were not told the specific characteristics of Scaladoc that were being studied.

Scala library method `List.filter`

• *Signature in "alpha" and "inherit" modes*:

```
def filter (p: (A) => Boolean): List[A]
```

• *Signature in "javadoc" mode*:

```
Inherited from trait TraversableLike[+A, +Repr]
...
def filter (p: (A) => Boolean): Repr
```

GData method `CalendarEventFeed.getEntries`

• *Signature in "alpha" and "inherit" modes*:

```
def getEntries(): List[CalendarEventEntry]
```

• *Signature in "javadoc" mode*:

```
Inherited from
  class BaseFeed[F <: BaseFeed, E <: BaseEntry]
...
def getEntries() : List[E]
```

Figure 1. Scaladoc signatures in different UI modes.

Sixty subjects completed the study, whose data was used for the analysis.

2.2 Task

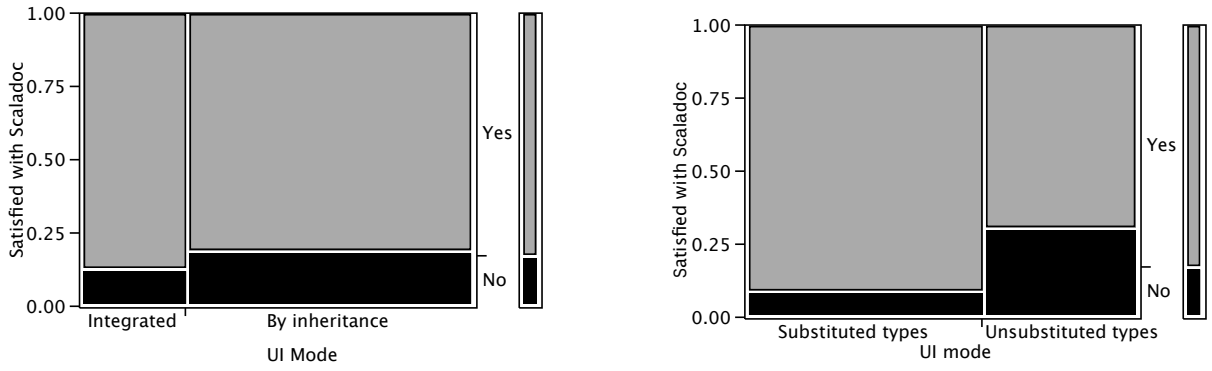
Subjects were divided into three groups, each receiving a different version of Scaladoc corresponding to the UI modes in Fig. 2. They were each asked 4 questions (some multipart) about the Google Data API and 4 questions (some multipart) about the Scala collections API. Questions are listed in Appendix A. Subjects' responses to questions as well as the time to answer the question were recorded. We intentionally chose both a Java API and a Scala API, so as to increase the applicability of the results.

Subjects were then asked a series of subjective questions about their experience with the user interface to assess their satisfaction with the documentation. The complete list of these questions appears in Appendix B. There were 3 sets of questions with a 5-point scale from "strongly disagree" to "strongly agree" and one yes/no question asking if their experience with Scaladoc in the study was overall satisfactory.

3. Results

3.1 Quantitative

Response time and correctness. Considering the sets of Google Data and Scala collections questions as a whole, there were no statistically significant differences between the UI versions for correctness of response or response time (of correct responses). The study contained several questions



(a) (b)

Figure 2. Effect of user interface parameters on user satisfaction. Figure (a) does not show a statistically significant result ($p = 0.45$), but figure (b) does ($p = 0.0333$).

concerning the subject's self-assessed programming skills, with the aim of using this data in the analysis to remove the variation due to skill between subjects. Upon analysis it appears that self-assessed programming skill did correlate with real performance. However, even after taking into account reported skill, the amount of unassigned variation in the results did not yield statistically significant differences between the UI versions.

It is possible to calculate the effect of UI version on each question separately, but this does not have a great deal of statistical power; since the 95% confidence applies to each test separately, with a large number of individual tests, we expect 5% of the tests to yield a "statistically significant" result by chance alone.

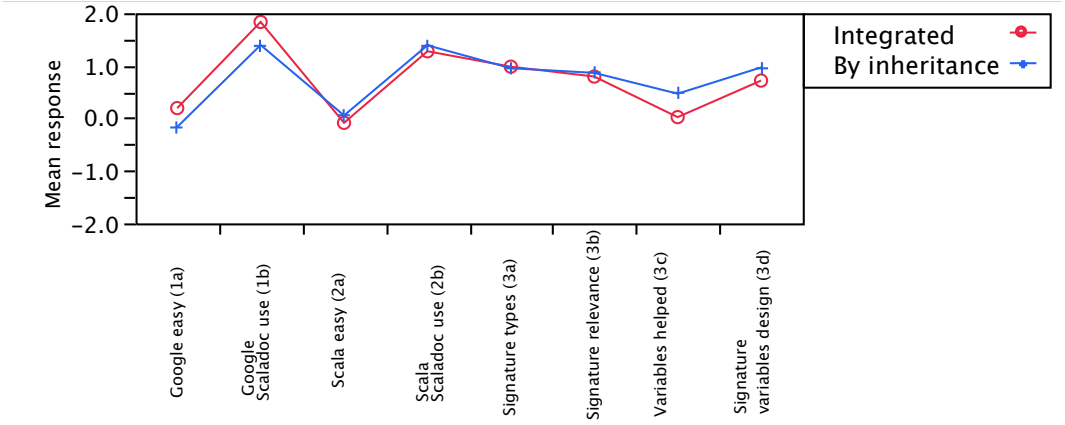
However, we found that one of the questions was essentially directly measuring the effect of instantiated type variables, without requiring any additional coding tasks. This particular question was the simplest of the Scala questions, and required that the subject look up a method signature for a method that was defined in a superclass and whose type changed if type parameters were shown in the context of the subclass. For this question, a t-test showed a statistically significant difference in mean response time, which was 35% higher ($prob > t = 0.0311$) for the user interface mode "javadoc" when compared to either "alpha" or "inherit." For the remaining Scala questions, we found that Scala skill was strongly correlated with time to correct response (using the logarithm of skill, ANOVA F- and p-values are respectively: 9.44 and 0.0038, 11.86 and 0.0013, 9.30 and 0.0050), which leads us to hypothesize that this factor masked the effect of the user interface mode.

Satisfaction. Regarding overall satisfaction with Scaladoc for the purposes of the study (the final question in the questionnaire), there was no statistically significant difference in user satisfaction ($p = 0.45$) in the integrated view as compared to the by-inheritance view (i.e., "alpha" as compared to "inherit" and to "javadoc"). On the other hand, subjects did significantly prefer (using Fisher's exact test for categorical data: $prob = 0.0333$) the user interface modes that substituted parent type variables (i.e., "alpha" and "inherit") to the mode that did not ("javadoc"). These results are summarized in Fig. 2.

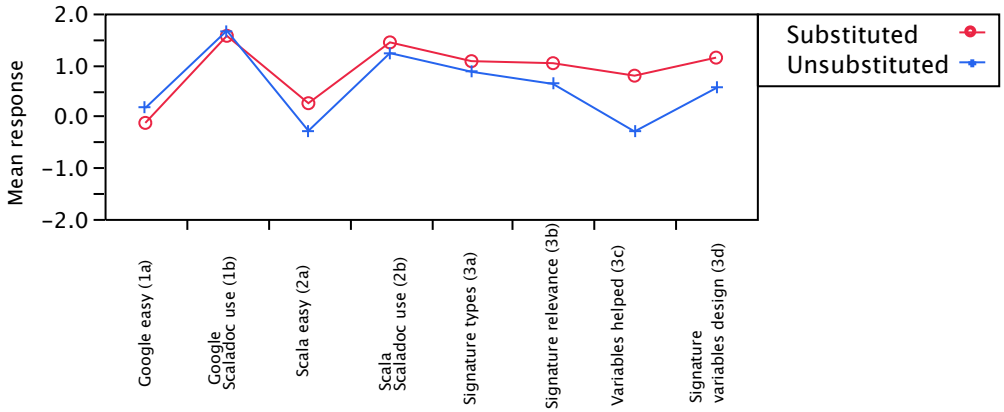
The level of satisfaction reported by subjects for the series of eight agree/disagree questions on detailed satisfaction-related items was analyzed using a MANOVA fit test. This test was chosen because the response variables (the answers to the satisfaction question) are obviously correlated. The factors considered in the analysis were the UI's inheritance and type display modes as well as a synthetic indicator of self-reported Scala skills and the self-reported Scaladoc 2 skill.

Figure 3 shows the mean response values for all eight response variables by UI display modes. It is already visible on these graphs that no different answering trend exists between the two UI inheritance display modes (Fig. 3(a)). On the other hand, the UI with substituted types earns generally higher scores than that with unsubstituted types. Interestingly, the results are consistently higher for the substituted UI when considering questions that relate to the satisfaction with the Scala library task or generally about Scaladoc, but are mixed when considering those that relate to the Google Data API task.

Reducing the response to a simple equal-weight sum of the response variables confirms the trend visible on the graphs: the UI's type display mode has a significant effect on the reduced response ($F = 0.1457$, $Prob > F = 0.0095$); all other factors have no significant effect. The data was further analyzed with a more refined MANOVA test, which models the answer to each question separately (using the identity matrix for transforming the response variables). The whole-model fit is significant (Pilai's trace test $F = 1.8369$, $Prob > F = 0.0002$). This test confirms the role of the UI's type display mode ($F = 0.4874$, $Prob > F = 0.0198$)



(a) Satisfaction results for integrated vs. by-inheritance view



(b) Satisfaction results for substituted vs. unsubstituted types

Figure 3. Satisfaction results for various UI parameters. The value (2) corresponds to "strongly agree" and (-2) corresponds to "strongly disagree." See Appendix B for the list of satisfaction questions.

and the absence of role of the UI's inheritance display mode ($F = 0.1650$, $Prob > F = 0.5353$).

On the other hand, this analysis shows that both skill-related factors have an effect on the responses. It is interesting to note that, while Scala self-reported skills have a positive influence on agreeing that the Google Data API questions were easy to answer, they have a slight negative influence when considering the Scala library questions.

3.2 Qualitative

We analyzed the freeform comments for anything directly related to the user interface variations that we used. We found that 1 out of 16 subjects who were given the "inherit" user interface and 4 of 18 users who used the "javadoc" user interface, wanted an alphabetical list of methods (rather than grouping based on parent class). Additionally, 4 of 18 users who used the "javadoc" user interface complained about types being shown in terms of parent type parameters, rather than instantiated types.

4. Discussion and Future Work

The subjects took longer to complete the study than we expected (mean time to completion was 30.5 minutes, median was 28.7 minutes), which suggests that the questions were difficult for many of the subjects. Making the tasks easier could make it easier to isolate the variables that we wished to study and could reduce the noise in the result data.

On a related note, we expect that it would be beneficial to repeat the study using a within-subject design. As mentioned above, we suspect that individual variation in skill, mood, and other uncontrolled factors had a large impact on the response time and correctness, so testing different user interfaces on the same subject may allow a better statistical control of these factors and yield stronger results. Of course, such a study would be a more serious undertaking, as more subjects would be needed in order to yield statistically significant results.

Another factor that may have increased the noise level in the data was that subjects took the study online, with

no control by the experimenters over their environment or degree of commitment to the task.

Broader Implications. This study shows that two language features, inheritance and generics, can have unexpected interactions when it comes to the problem of displaying documentation. Both of these features are important and can improve the quality of code. But, documentation generation tools did not, until now, consider the impact of these features on documentation.

The lesson for tool developers, be it Javadoc-like generators or IDEs, is that a new language feature can have a far-reaching impact on the information needs of programmers. This study only scratches the surface of how documentation can be improved. Corpus studies such as that in Table 1 seem a promising avenue for finding new directions in which tools can be improved.

5. Related Work

Few researchers have studied the design of Javadoc itself, aside from the studies that led to its initial design [2]. The specific problems of Javadoc that we have considered seem to have been largely ignored by most documentation tools and IDEs.

For example, when performing a dot-completion, the three top Java IDEs, Eclipse, IntelliJ, and NetBeans, all show a flattened view of all members of a particular object with correctly instantiated type parameters; however, in the outline view for a particular class, parent type parameters are not instantiated. In C#, only the Visual Studio Object Browser shows correctly substituted type parameters. Other IDE tools, such as the JetBrains ReSharper plugin[2] do not show instantiated type parameters. Additionally, the C# documentation generator does not show types in its summary view, only method names and short comments. Clicking on an inherited method will show the definition in the parent class, with parent type parameters.

Instantiating parent type parameters bears some similarity to functor application in ML. However, documentation generators such as OCamldoc[3] do not show instantiated types for modules that are the result of functor application, even though the OCaml REPL does show the expanded type. That is, in the generated documentation, the set of members of the module are shown in terms of the types of the functor's formal, rather than actual, parameter.

References

[1] Google. Google Data Java client library. Available at `http://code.google.com/p/gdata-java-client/`, 2010.

[2] D. Kramer. API documentation from source code comments: a case study of Javadoc. In *SIGDOC '99: Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM. ISBN 1-58113-072-4. doi: http://doi.acm.org/10.1145/318372.318577.

[3] C. Marshall. Is the Scala 2.8 collections library a case of "the longest suicide note in history"? Available at http://stackoverflow.com/questions/1722726/is-the-scala-2-8-collections-library-a-case-of-the-longest-suicide-note-in-histo, 2009.

[4] M. Odersky. Scala 2.8 collections. Available at `http://www.scala-lang.org/sid/3`, 2010.

[5] M. Odersky and A. Moors. Fighting bit rot with types (experience report: Scala collections). In R. Kannan and K. N. Kumar, editors, *FSTTCS*, volume 4 of *LIPIcs*, pages 427–451. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2009. ISBN 978-3-939897-13-2.

[2] `http://www.jetbrains.com/resharper/index.html`

[3] `http://caml.inria.fr/pub/docs/manual-ocaml/manual029.html`

A. Task Questions

The main task comprised the following questions, presented to users in all groups.

Question 1

```
import com.google.gdata.client.calendar._
import com.google.gdata.data.calendar._
import com.google.gdata.data._

/** Update the title of the entry */
def updateTitle(entry: CalendarEventEntry,
    newTitle: TextConstruct
   ): /* TODO 1: fill in return type */ = {
  // TODO 2: update the entry title
  entry.update
}
```

- What is the return type of the method "updateTitle"?
- Insert the code marked by the comment "TODO 2"

Question 2

```
import com.google.gdata.client.calendar._

/** Get e-mail addresses of each participant in the event */
def getParticipantEmails(
     entry: CalendarEventEntry): Iterable[String] = {
  // TODO: return an iterable of participant e-mail addresses
}
```

- Fill in the code for the TODO.

Question 3

```
import com.google.gdata.client.calendar._
import com.google.gdata.data.calendar._
import java.net.URL

/** Add calendar with name 'calendarName' into feed */
def addCalendar(service: CalendarService,
     calendarName: String, calendarsUrl: URL
   ): /* TODO 1: insert type */ = {
  val calendar = new CalendarEntry
  calendar.setTitle(new PlainTextConstruct(calendarName))
  // TODO 2: insert the calendar into the feed indicated by 'calendarsUrl'
}
```

- What is the return type of the method "addCalendar"?
- Insert the code marked by the comment "TODO 2"

Question 4

```
import com.google.gdata.client.calendar._

/** Print all events in 'theFeed' */
def printAllEvents(theFeed: CalendarEventFeed) = {
  // TODO: print the title of each event in 'theFeed'
}
```

- Insert the code marked by the comment "TODO"

Question 5

```
def clup(cs: immutable.List[Char]): Unit = {
  val csg: /* TODO: insert type */ = cs.groupBy({
   case c if c.isWhitespace => ' '
   case c if c.isLower => 'l'
   case c if c.isUpper => 'U'
   case _ => '_'
  })
}
```

- Insert a type for 'csg', as marked by the comment "TODO".

Question 6

```
val csgm = mutable.Map.empty[Char, mutable.Buffer[Char]]

for ((key, value) <- csg) {
  csgm.update(key, mutable.Buffer.empty[Char])
  // TODO: Add the elements in 'value' to the empty
  // buffer just added in 'csgm' at 'key'
}
```

- Fill in the code marked by TODO.

Question 7

```
val vowels =
  mutable.Buffer('a', 'e', 'i', 'o', 'u', 'y')
// TODO: Update 'vowels' to add upper case vowels
// by mapping over the lower case vowels (use toUpper)
```

- Fill in the code marked by TODO.

Question 8

```
val csgm:
  mutable.Map[Char, mutable.Buffer[Char]] = ...

/* insert argument and return type.
   look at use of function in definition of 'csk' below */
def removeVowels(
    chars: /* TODO 1: insert arg type */
   ): /* TODO 2: insert return type */ = {
  chars.diff(vowels)
}

val csk: /* TODO 3: insert type of csk */ =
  csgm.keySet
  .filter({ c => c.isLetter })
  .map({ c => removeVowels(csgm(c)) })
```

- Give the type of the argument to 'returnVowels', as indicated by TODO 1.
- Give the return type of 'returnVowels', as indicated by TODO 2.
- Give the type of 'csk', as indicated by TODO 3.

B. Satisfaction Questionnaire

The following questionnaire was presented to subjects after they had complete the main task questions. For questions 1–3, subjects were presented with a 5-point scale of "strongly disagree," "disagree," "neutral," "agree," and "strongly agree." Question 4 required a yes/no response.

1. Say whether you agree with the following statements concerning the questions on the Google Data Client library (GData):
   (a) I found the GData questions easy to answer.
   (b) The Scaladoc played an important role in my answering the GData questions.
   (c) I have already written programs that contain code like that found in the GData questions.
2. Say whether you agree with the following statements concerning the questions on the Scala collection library:
   (a) I found the collection questions easy to answer.
   (b) The Scaladoc played an important role in my answering the collection questions.

(c) I have already written programs that contain code like that found in the collection questions.

3. A member is a method, value, variable or any other element owner by a class or object. Its signature contains its parameter types (if it is a method) and return type. For example, the following is the signature of method `union` in `List`:

   ```
   def union (that: Seq[A]) : List[A]
   ```

   Say whether you agree with the following statements concerning the member signatures displayed in Scaladoc:

   (a) Member signatures were those I expected to see.

   (b) I could relate the member signatures in the documentation to the task at hand.

   (c) Type variables in member signatures helped me solve the task.

   (d) Type variables im member signatures help me better understand the design of the API.

4. Were you satisfied with Scaladoc during this study?