Understanding the Problem of API Usability and Correctness Misalignment

Tao Dong
Google LLC
Mountain View, CA, USA
taodong@google.com

Elizabeth F. Churchill
Google LLC
Mountain View, CA, USA
echurchill@google.com

Abstract
When developers have more than one API they could potentially use to solve a programming problem, it's often natural for them to start with the easier and familiar, often default, option. Yet, for some tasks, such as manipulating text in the presence of grapheme clusters (e.g., g̈ and 한), the easier API could produce less correct and reliable results. We sought to measure the impact of such misalignment between API usability and correctness. Specifically, we conducted a controlled experiment which shows that user education has a limited effect on helping the programmer choose the appropriate API, when it's not the default and error cases are difficult to imagine. We discuss a few things programming language and SDK designers can consider in order to mitigate the impact of such misalignments.

1. Introduction
API usability researchers (Myers & Stylos, 2016) have shown that poor usability of APIs can lead to flaws in computer programs, and sometimes serious security flaws (Fahl et al., 2013). However, when there are multiple APIs the programmer can choose from to solve a given problem, using the most usable API does not necessarily lead to the expected outcome. What if the easier to use API is in fact more likely to produce incorrect or unreliable results under some circumstances? We call this problem "API usability and correctness misalignment." We believe this is a challenge that has been under-examined by API usability researchers.

One of the areas where such a misalignment problem has manifested itself is text manipulation, specifically handling grapheme clusters in unicode strings. In many programming languages, text is often represented as a sequence of UTF-16 code units. However, some user-perceived characters need to be backed by two or more UTF-16 code units. In fact, this is quite common once you need anything beyond the ASCII character table. For example, the letter g̈ is composed of a base character g and a combining mark ̈, the Hangul syllable 한 may be represented by a sequence of conjoining jamos ᄒ, ᅡ, and ᆫ, and the emoji 👋 is encoded as a *surrogate pair* (Microsoft, 2018) of two UTF-16 code units 55357 and 56395. In the Unicode Standard, such user-perceived characters are formally known as grapheme clusters (Davis & Chapman, 2020).

Grapheme clusters can easily break text manipulation code written without anticipating their presence, because the default String APIs in most popular programming languages are not aware of them, and they instead operate on fixed-length code units such as UTF-16. For instance, the following line of code in Dart will produce an unexpected result:

```
print('Hello👋, world!'.substring(0,6));
```

The programmer's intent here is to print the substring "Hello👋." However, the end index "6" (exclusive) will include "Hello" and then the first half of the surrogate pair representing the 👋 emoji. As a result, this line of code will return a substring with a malformed trailing character: "Hello�." While this particular example is from the Dart programming language, the problem is shared by many others. One notable exception is Swift, a more recently launched programming language, first released in 2014.

For mature programming languages such as Java, Python, and Dart, a common approach to adding support for correctly processing grapheme clusters is providing an auxiliary grapheme-aware text manipulation library while keeping the default String API intact for backwards compatibility and performance reasons. Nonetheless, this seemingly practical and sensible solution also leads to a misalignment between usability and correctness, forcing the programmer to choose between multiple text manipulation APIs with hard to test real-world implications. To languages that have taken this path to supporting grapheme clusters, the easiest to use text manipulation API is often the built-in String API due to its availability, familiarity, and accumulated documentation and community knowledge. However, it's often the less correct API to use, especially when the programmer cannot limit the kinds of characters included in the text they manipulate, as it's often the case when the text is originated from user input.

To understand the severity of such misalignment between usability and correctness in making API choices, we conducted a web-based controlled experiment in the context of text manipulation in the Dart programming language. Dart recently added a package called *characters* to support grapheme clusters while keeping its UTF16-based String API unchanged. In our experiment, participants were asked to examine a number of code snippets, written with Dart's String API, for common text manipulation scenarios and determine whether each snippet would produce the expected results. Our analysis of the experiment results led to three main findings:

- Prior exposure to the grapheme clusters problem did not help participants identify it in text manipulation code.
- Participants in the two treatment groups who received information about the grapheme clusters problem and the characters package early in the experiment, showed a significant improvement in their ability to make correct assessments of text manipulation code over those in the control group who did not receive such information.
- Nonetheless, more than half of the participants who received an explanation about the grapheme clusters problem still failed to apply that knowledge to assessing code snippets.

The results demonstrate a substantial limit of user education in both the durability and strength of its effect, when the usability and the correctness of API choices are misaligned and no immediate feedback could be provided to the programmer. Our findings suggest that modernizing the default String API to support grapheme clusters should be seriously considered by programming language designers, especially when the programming language is not yet widely used and can afford making breaking changes to the String API.

Nonetheless, when overhauling the default String API is infeasible, language and SDK designers should mitigate the usability and correctness misalignment by facilitating the programmer's choice making process. There are a few ways to achieve it beyond user education:

- Making the alternate API more readily available and frequently visible to the programmer through a tight integration with the programming environment.
- Making the alternate API a local default through proactive suggestions and warnings in coding contexts where the risk of choosing the wrong API is high.
- Helping programmers write test cases that cover edge cases where the alternate API should be used rather than the default API.

The rest of the paper is organized as follows. First, we review related work on API usability and choice architecture. Next, we go over the design of the experiment and main findings from analyzing the results. Last, we discuss possible explanations for the experiment results and the implications for programming environment design.

2. Related Work

We contextualize our work at the intersection of API usability and the psychology of making choices. Due to space limits, we describe most relevant work in these two areas and explain the intellectual gap our research addresses.

2.1. API Usability

The importance of API usability has started gaining recognition in the industry after more than a decade of advocacy by API usability researchers (Clarke, 2004). Myers and Stylos (2016) argue that unusable APIs can lead to bugs, performance degradation, and even significant security problems. They suggest adopting the human-centered design approach in the process of designing APIs. This approach calls for evaluation of API design in accordance with various usability principles such as those adapted from Nielsen's "heuristic evaluation" guidelines (Nielsen, 1994) and cognitive dimensions of notations (Blackwell et al., 2001).

Nonetheless, the challenge of making API choices is under-examined. The main paradigm in API usability research has been focused on the design of a single API or a single set of APIs intended to be used together. The common approach of API usability research usually involves observing how programmers learn and use the API in question, identifying user experience problems, and making recommendations on how to improve the API's design and documentation (for example, Piccioni et al. (2013)). Some studies compared multiple designs of an API in order to select a more usable one (Stylos et al., 2006). In a rare instance, researchers paid attention to the problem of choosing from multiple APIs for the same purpose and found that the presence of two APIs with similar names caused widespread programmer confusion (Murphy-Hill et al., 2018).

Few studies have examined the programmer's ability to make API choices, especially when those choices involve subtle trade-offs between usability and other goals of API design, such as correctness, backwards compatibility, and computational efficiency (Stylos & Myers, 2007). The prevailing paradigm in API usability research often implicitly considers those other goals as variables independent from the API's usability characteristics. While this is a useful simplification, it does not sufficiently recognize the inherent tensions and dependencies between some of those API design goals. For example, a high-level API is often more usable, but it could lack flexibility or coverage for edge cases. How well can programmers understand and evaluate such tradeoffs? And how can API and tooling designers provide appropriate signifiers and constraints to support the programmer in making API choice decisions? We believe it is important to extend the focus of API research from a single solution to all solutions the programmer can choose from in order to satisfy a programming requirement. This shift of focal point is critical to address the complexity of evolving APIs already widely used in production.

2.2. Choice Architecture

There has been much research in the area of human decision-making and option selection in the face of alternatives. This domain of investigation is explicitly about designing choices, hence the name "choice architecture design," coined by Thaler and Sunstein (2009). Our particular interest is in the ways in which API choices are presented to programmers, and whether it is possible to understand the ways in which defaults explicitly presented or tacitly assumed can lead to poor choices that have negative downstream consequences for users of the program in question.

It is well known from studies of choice architectures and user interface design that people usually stick with defaults, unless there are clear indications of costs and benefits of making alternative choices. Schneider et al. (2018) have shown that UI design influences choices, even unintentionally; they state "user interface, from organizational website to mobile app, can thus be viewed as a digital choice environment" nudging people to take certain actions over other by modifying what is presented or how it is presented. The example they cite is the Square mobile payment app which presents a "tipping" option by default; customers must select "no tipping" if they prefer not to give a tip which is additional effort and also triggers social "norming" and social belonging.

With respect to defaults specifically, Jachimowicz et al. (2019) in a meta analysis suggest two factors that partially account for the variability in defaults' effectiveness: the contexts and domains where defaults were presented and their relative ease of implementation. Their findings also point to the importance of evaluating the intended population's preferences when deciding when and how to deploy defaults. Huh et al. (2014) show that the observed choices of others can become social defaults, increasing their choice share. Social default effects are a novel form of social influence not due to normative or informational influence: participants were more likely to mimic observed choices when choosing in private than in public and when stakes were low rather than high.

The empirically observed difficulty of making rational choices, under the influence of defaults in particular and other choice architecture interventions in general, has been subject to several different explanations about human cognition and reasoning. One of those explanations invokes the notion of *bounded rationality*, in which Simon (1996) argues that the decision maker is often limited by the availability of information and their cognitive ability to process all the information pertinent to the decision in order to make the optimal choice. As a result, the decision maker would "satisfice" – settling for a reasonable choice but not necessarily the best one. Another explanation points to the *failure of imagination*. Shackle (Shackle, 1964) contends that "Choice is amongst imagined experiences," and Augier (2000), elaborating on Shackle's theory, suggests that "Choice is amongst imagined experiences," and "each alternative is considered with a variable amount of disbelief." This point of view is particularly relevant to decision making in software engineering, as Somers (2017) has warned about the challenge of reasoning about program behavior in complex software systems.

It is worth noting that the aforementioned assignment of disbelief to different imagined scenarios is likely a process adaptive to subtle cues available in the decision-making context. Through a number of laboratory experiments, McKenzie at al. (McKenzie et al., 2018) demonstrate that a number of apparent "biases" in making choices stem from adaptive sensitivity to subtle contextual cues in the choice environment, which dynamically update the decision maker's belief about the desirable course of action or the attribute distributions in the population. For example, the default option is often perceived as the recommendation from someone in a position of authority.

This perspective on dynamic belief and preference construction not only provides a different view of the psychology and rationality of decision making, it also suggests a different approach to choice architecture design. Whereas the traditional nudge approach tries to engineer specific decision outcomes, often by rerouting apparent biases so that they point in desirable directions, the authors suggest an alternate approach focused on facilitating the processes of decision making. For instance, McKenzie at al. (2018) called for overt messaging instead of covert "nudges" to support decision makers by increasing information salience in accordance to the information's relevance and the decision maker's attentional capacity. The above perspectives and results have informed our problem framing, interpretation of data, and development of mitigations for the API usability and correctness misalignment problem we examined.

3. Study Design

3.1. Overview

To measure the impact of the API usability and correctness misalignment, we conducted a web-based controlled experiment. In particular, we wanted to check how much of the problem can be mitigated by getting the programmer to read documentation about the problem, since API designers often resort to "user education" as the first and easiest to implement response to issues related to user experience.

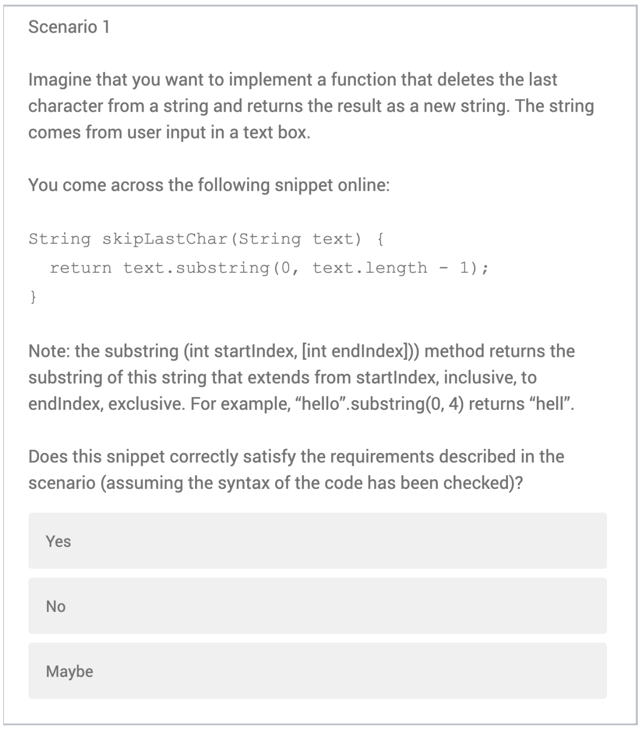
The experiment was implemented as a scenario-based questionnaire using the survey software Qualtrics and administered over the Internet. The questionnaire had four main parts:

- **Screener**: Participants of the experiment were recruited from Dart's user community without incentives. At the beginning of the experiment, each participant answered a few questions about their background and their knowledge about Dart in order to determine their eligibility.
- **Code snippet review tasks**: This was the main body of the questionnaire. The participant was asked to assess the correctness of code snippets written for six common text manipulation scenarios (see subsection 3.3). Participants in the treatment groups (more about experimental conditions below) received information about the grapheme clusters problem in the Dart String API after Scenario 1, simulating an experience of getting educated about the problem.
- **Reflect on assessment results**: After reviewing the code snippets in those six scenarios, participants in the treatment groups were given an opportunity to review and reflect on the correct assessments of the code snippets they examined.
- **Post-test questionnaire**: In the last part of the experiment, participants answered questions about their attitudes, preferences, and prior experience that might help contextualize their responses.

The full questionnaire design is available in the Appendix.

3.2. Experimental conditions

The experiment had a control condition and two treatment conditions. We randomly assigned the 183 participants to one of the three experimental conditions. The three conditions primarily differed in the amount of information about the grapheme clusters problem given to the participant after they evaluated the code snippet in Scenario 1, which was about extracting a substring (shown in Fig. 1).



Scenario 1

Imagine that you want to implement a function that deletes the last character from a string and returns the result as a new string. The string comes from user input in a text box.

You come across the following snippet online:

```
String skipLastChar(String text) {
  return text.substring(0, text.length - 1);
}
```

Note: the substring (int startIndex, [int endIndex])) method returns the substring of this string that extends from startIndex, inclusive, to endIndex, exclusive. For example, "hello".substring(0, 4) returns "hell".

Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)?

Yes

No

Maybe

Figure 1 - The first of the six code assessment scenarios participants went through in the experiment. The code snippet could produce incorrect results when the input text had grapheme clusters.

Specifically, the control group received no feedback at all after assessing scenario 1 and continued to review the rest of the code snippets. Both treatment groups received basic information about the fact that the Dart String API could produce incorrect results when extracting a substring from text that included grapheme clusters and an example of revising the snippet using the characters package. Treatment group 1 was given further explanation about the underlying UTF-16 representation of text in Dart's default String API and why the API won't handle grapheme clusters correctly. Table 1 summarizes the differences between the three experimental conditions.

Experimental Condition	Explanation Received after Scenario 1
Control	No explanation
Treatment 1	Full explanation
Treatment 2	Light explanation

Table 1 – The differences between the three experimental conditions.

3.3. Experimental tasks

As aforementioned, each participant was asked to review code snippets written for the following six text manipulation scenarios (see an example in Fig. 1):

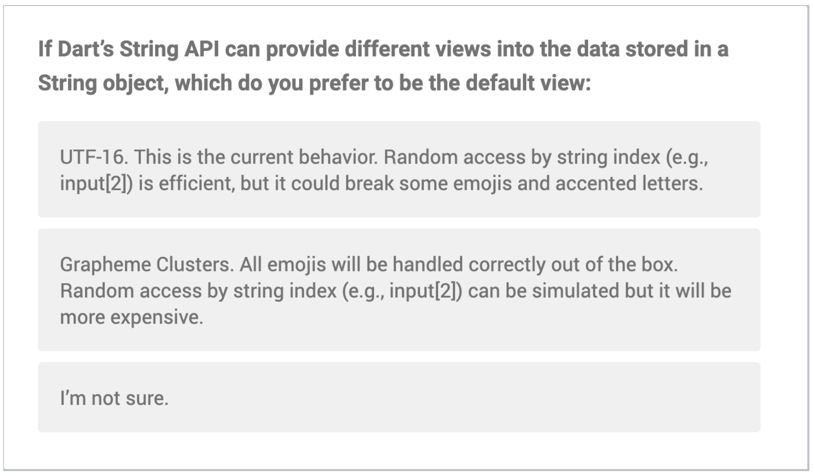
1) Extracting a substring
2) Validating email addresses using a regular expression
3) Checking character limit
4) Splitting a string on an emoji
5) Creating initials from a first name and a last name

6) Turning overflown text into an ellipsis

The code snippets in the 6 scenarios all used the Dart String API to process their respective text input. Among them, Scenario 1, 3, 5, and 6 would produce unexpected results when grapheme clusters were part of the text being manipulated. Programmers could use the characters package instead as a remedy. The experiment used those scenarios to check false negatives – failures to identify potential programming errors. In contrast, Scenario 2 and Scenario 4 were designed to check false positives – the potential of participants overacting to the information provided about grapheme clusters and hence mistakenly dismissing the correct use of String API in those two scenarios.

3.4. Post-test questionnaire

The post-test questionnaire had two pages. The first page was focused on the participant's attitudes towards the grapheme clusters problem and their preferred default string data representation (see Fig. 2). This section of the questionnaire was only displayed to participants in the treatment groups, since the control group was not informed of the grapheme clusters problem in the experiment.



If Dart's String API can provide different views into the data stored in a String object, which do you prefer to be the default view:

UTF-16. This is the current behavior. Random access by string index (e.g., input[2]) is efficient, but it could break some emojis and accented letters.

Grapheme Clusters. All emojis will be handled correctly out of the box. Random access by string index (e.g., input[2]) can be simulated but it will be more expensive.

I'm not sure.

Figure 2 - One of the questions in the post-test questionnaire is about preferred string representation in Dart's String API.

On the second page of the questionnaire, the participant was asked to provide their background information, such as awareness of the grapheme clusters problem, prior knowledge about Dart's characters package, and familiarity with Swift's String API, which might help explain their behavior and attitudes shown in the experiment.

3.5. Hypotheses and data analysis

As mentioned, part of the experimental goal was to measure how much we can rely on programmer education (e.g., reading documentation) to help the programmer make sound API choices and hence mitigate the impact of the usability and correctness misalignment. Thus, the experiment was designed to test the following hypotheses:

- H1: Participants who had prior exposure to the grapheme clusters problem in text manipulation were more likely to identify the problem in code snippets.
- H2: Participants who were informed of the grapheme clusters problem early in the experiment (i.e., the two treatment groups) would be able to more accurately assess the correctness of the code snippets than those who weren't (i.e., the control group).
- H3: Participants who were given more in-depth explanations of the grapheme clusters problem early in the experiment (i.e., treatment group 1) would do better in identifying the problem in the code snippets than those given a basic explanation (i.e., treatment group 2).
- H4: The majority of participants would prefer a String API aware of grapheme clusters by default after reflecting on their own ability to make correct API choices.

4. Findings

In this section, we report the main findings from analyzing the results of the experiment by the order of the hypotheses stated above. Due to space limits, we omit the results about false positives in evaluating Scenario 2 and 4. Assessment results of Scenario 2 were also muddled by participants' lack of experience with regular expressions.

4.1. Prior exposure to the grapheme clusters problem showed a negligible effect

As expected, a minority of our 183 participants reported prior awareness of the grapheme clusters problem. Specifically, 53 said they had only heard of the issue, and 22 said they had run into the issue themselves. The analysis below was focused on participants' assessment of the code snippet in Scenario 1, before participants in the two treatment groups were informed of the grapheme clusters problem after this scenario, as described in the Study Design section. As a reminder, the correct response to the assessment question in Scenario 1 should be "No."

Among the 53 participants who had heard of the issue before, 26% made a correct assessment of the code snippet in Scenario 1. And among the 23 participants who claimed to have first-hand experience with the problem, 23% made a correct assessment. Both were only slightly better than the participants who reported no knowledge of the issue prior to assessing Scenario 1, as shown in the green portions of the bar chart in Fig. 3. A chi-squared test didn't show statistical significance in the differences between those without prior knowledge and those who were aware of the problem to some degree. Therefore, the data does not support H1.

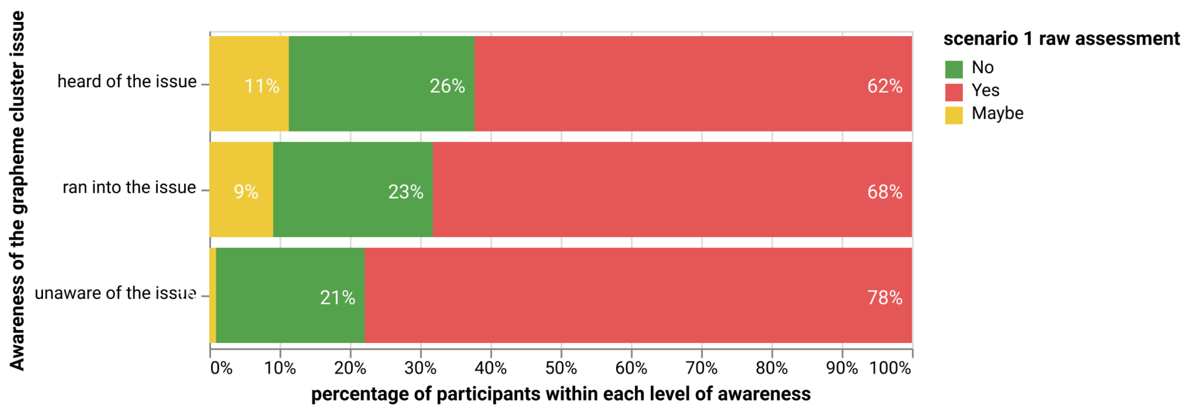


Figure 3 - Participants' assessments of the code snippet in Scenario 1 broken down by their prior awareness of the grapheme clusters problem. The correct answer should be "No."

4.2. Explaining the grapheme clusters problem upfront was useful but insufficient

In the experiment, both treatment groups were informed of the correct assessment of the code snippet in Scenario 1 and given different amounts of explanation about why the snippet could produce incorrect results. Since this information was given right before those participants went on assessing the rest of the code snippets, it's not surprising that they performed much better than the control group (see Fig. 4). For example, in Scenario 3 (counting the number of characters), only 15.5% of the participants in the control group were able to determine that the snippet could produce wrong results when grapheme clusters were part of the string, while 46.3% of treatment group 1 and 36.2% of treatment group 2 were able to do so. Moreover, chi-squared tests showed statistically significant differences across the three groups' assessment performance for the three scenarios (#2, #5, and #6), where the provided code snippets could have trouble processing grapheme clusters.

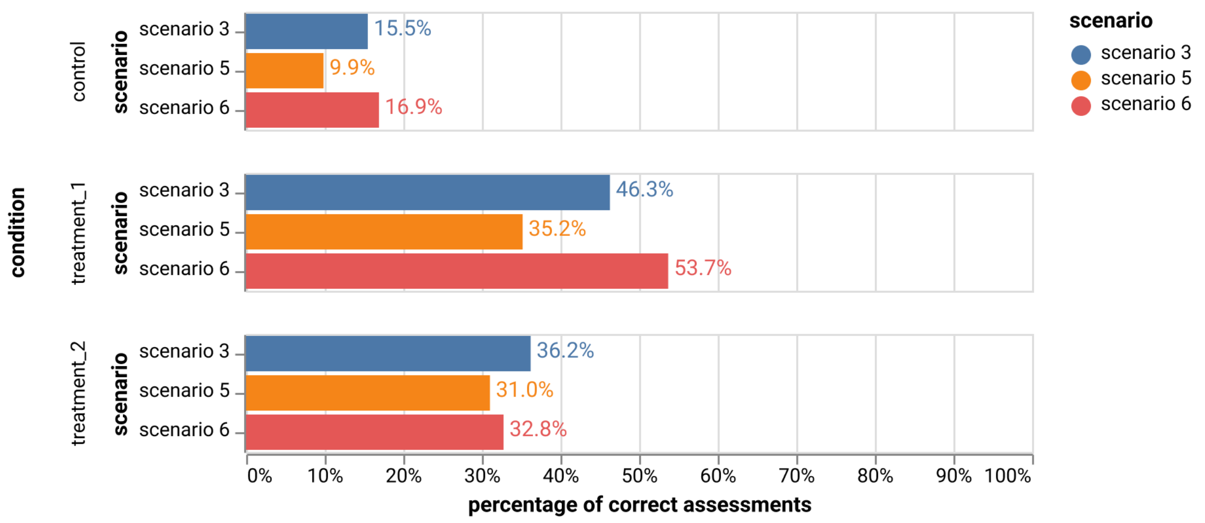


Figure 4 - The percentage of participants who made a correct assessment of the code snippet in Scenario 3, 5, and 6 across three experimental conditions

Disappointingly, both treatment groups still made more incorrect assessments than correct assessments. On average, treatment group 1 (received an in-depth explanation about the grapheme clusters problem) and treatment group 2 (received a basic explanation) achieved an average assessment success rate of 45% and 33%, respectively. In conclusion, the data does support H2, but the effect of providing the participants with the documentation of the grapheme cluster problem and the characters package is much weaker than many might have expected.

4.3. The amount of explanation only made a small difference

In the experimental design, we gave treatment group 1 a deeper and more thorough explanation in the hope that participants in this group could apply this knowledge to the assessment of subsequent text manipulation scenarios that were different from Scenario 1. The data shows that treatment group 1 did seem to perform better than treatment group 2 in every scenario where using the String API was problematic (see Fig. 4). The largest difference was observed in Scenario 6, where the code snippet needs to enforce a character limit in a text field and replace any overflown text with an ellipses. In this scenario, 53.7% of participants in treatment group 1 made a correct assessment, while only 32.8% of the treatment group 2 were able to do so. However, the difference between the two groups' performance in Scenario 6 was not statistically significant according to a chi-squared test ($p = 0.055$), so were the test results for the other two scenarios.

4.4. Participants lacked confidence in their ability to make correct API choices

As mentioned in the study design, participants in the treatment groups were given an opportunity to compare their own assessments of the coding scenarios with the correct assessments and read a brief explanation about each scenario. We were curious about how that opportunity to reflect on their assessment performance might influence their opinions about how the String API should be designed.

Responding to a multiple choice question in the post-test questionnaire, about half of those participants said they would prefer a string representation aware of grapheme clusters by default, given the trade off between correctness and a potential hit to computational efficiency (see Fig. 5).

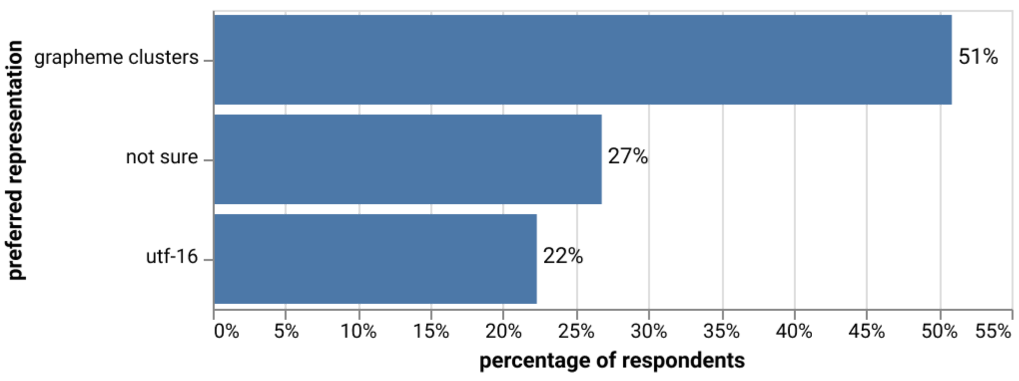


Figure 5 - Participants' preferred default string representation as stated in the post-test questionnaire

Some participants voiced concerns over their ability to be constantly on the lookout for subtle issues due to mishandling of grapheme clusters:

> *"Even when you understand how this currently works in theory, it is EXTREMELY easy to forget in practise. The current behavior WILL create many, many bugs."*

> *"...making the api catch those errors by default will improve the quality of apps since most developers don't handle emoji use cases until they have a real crash."*

For the minority who preferred UTF-16 as the default string representation (i.e., the status quo in Dart), some felt there was value to be consistent with other languages:

> *"Current behaviour is consistent with other language paradigms which allow low level manipulations."*

Others thought emojis were still rare in their particular use cases:

> *"In many use-cases, emojis probably won't show up. Let's say I'm developing an application for an insurance company, or maybe a medical info app. In these cases, it's highly unlikely to have emojis due to the professional nature of the app."*

To sum up, the data weakly supports H4, but a lot of participants were unable to make up their mind, reflecting the complexity of the trade-offs involved in modernizing a foundational API such as String in a mature programming language.

5. Discussion

In this section, we discuss what might have contributed to the difficulty of making API choices observed in the experiment, the implications for designing programming tools and environments, and the larger question of how priorities and values are framed and acted upon in software development.

5.1. Factors exacerbating the impact of misalignment

Our findings show that the misalignment between API usability and correctness can be difficult for the API user to deal with. The conventional wisdom might lead us down a path of more user education about the problem through documentation, but the results of our experiment, which in many ways simulated such an approach, suggest that it is insufficient. So why did our study participants, after they had been exposed to the grapheme clusters problem either before the study or during it, still experienced so much difficulty identifying it in the simple code snippets they examined? There could be several contributing factors, some of which we have touched on when describing related work:

- *Insufficient absorption and recall of information.* Although the explanation about the grapheme clusters problem and the characters package was provided to all participants in the treatment groups, there was no guarantee that every participant was able or willing to fully absorb this information in the first place and to recall that information later when it was needed in evaluating additional scenarios. This was essentially *bounded rationality* at work.

- *The failure of imagination.* It could be difficult to imagine an edge case that would produce incorrect results. Grapheme clusters is a complicated concept in unicode. Only some emojis and non-ASCII characters need to be represented in more than one UTF-16 code unit, and only specific types of text manipulations would break them. As a result, it's hard for programmers to come up with example strings to verify the correctness of their code.
- *The lack of learning transfer* (Perkins et al., 1992). Some participants might have understood why the first coding scenario was problematic and the grapheme cluster issue in abstract after receiving feedback, but they might not have developed the ability to apply this knowledge to the assessment of subsequent coding scenarios.
- *The habitual use of APIs and routinized operations***.** Because manipulating text is an extremely common task in programming, the ways of solving different text manipulation problems are likely to have become routines if not habits. It is thus difficult to critically evaluate familiar code snippets used so many times before without issues.

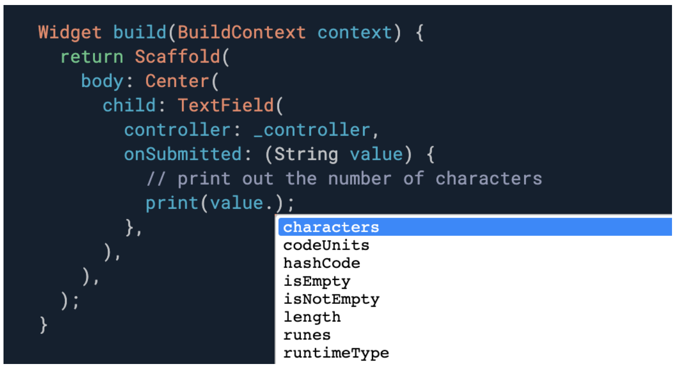
When one or more of these factors are at play, the impact of the misalignment between API usability and correctness can be especially hard to overcome.

5.2. Implications for design

While our results suggest that it is important to avoid or remove this type of misalignment in API design whenever possible, it is not always feasible when evolving an existing SDK with many users and real-world applications built on top of it. Forcing a realignment of the API correctness and usability could break backwards compatibility and lead to ecosystem fragmentation.

Nonetheless, the SDK designer could still consider the following mitigations that specifically address the three factors contributing to the difficulty of identifying and managing the misalignment.

- *Making the choices visible and readily available.* For example, Flutter, a popular UI framework for Dart, has integrated the characters package in its SDK, so Flutter programmers don't need to manually import it to their projects. In addition, the Characters API is also made available on String objects and literals through extension methods. Therefore, in a Flutter UI programming context, it's much easier to get reminded of this new, more robust way of manipulating text through the IDE's autocomplete facility (see Fig. 6).



```
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: TextField(
        controller: _controller,
        onSubmitted: (String value) {
          // print out the number of characters
          print(value.);
        },
      ),
    ),
  );
}
```

Figure 6 - Discovering the characters API is much easier after Flutter integrated it into its SDK.

- *Providing assistance in creating useful test cases.* This strategy intends to address the failure of imagination in making API choices. Specifically, programming tools need to help programmers write test cases that cover grapheme clusters by suggesting example input strings and showing warnings when existing test cases fail to cover edge cases. In addition to augmenting automated tests, a tool simulating diverse text inputs, such as emojis and non-ASCII characters, could help the developer discover text manipulation bugs early.
- *Breaking habitual use of API by making new local defaults.* When it's infeasible to change the global default, it might be possible to enact local defaults to mitigate risks of misusing APIs. For example, a UI toolkit often requires the programmer to create a callback function to handle user input. This particular context is highly susceptible to issues related to grapheme

clusters. The code editor could proactively complete a template for such callback functions where the grapheme-aware API is used in the generated code. Additionally, the code editor can use lints to warn programmers in potential high-risk text manipulation contexts.

5.3. Developer incentives and values

The need for processing grapheme clusters far predates emojis. In fact, they have been used to encode non-ASCII characters in many natural languages for almost 20 years (Davis, 2001). But why did this problem, a prime example of API usability and correctness misalignment, seem to draw little interest from the programming language design community until recently?

To answer this question, we need to critically examine software developers' incentives and the economics of designing and building software for a diverse user population. The value of being correct in the presence of grapheme clusters in text manipulation was probably considered less important than providing a straightforward, familiar, and efficient String API until two underlying economic forces became more prominent in the last few years:

- The rapid growth of both the proportion of Internet users who speak a language other than English and the proportion of non-English web pages (Wikipedia contributors, 2020)
- The rise of emojis in daily electronic communications (Danesi, 2016)

More recently designed programming languages, such as Swift, made the grapheme clusters a first-class citizen in its String API. This change of attitude mirrors the gradual rise of accessibility, which also suffered from *the failure of imagination*. For example, an able-bodied web developer could have trouble thinking through the experience of the site from the perspective of a visually impaired user. Modern developer tools provide facilities to simulate the experience of users with accessibility needs. We believe a similar approach can be fruitful in helping developers build awareness and empathy towards users with diverse text input needs.

6. Conclusions

In this paper, we proposed and examined the problem of misaligned API usability and correctness – the more usable API produces less correct results than a harder to use or less familiar API. We measured how well programmers can handle this type of misalignment through a controlled experiment in the context of manipulating unicode text that includes grapheme clusters, widely used to represent emojis and characters in non-English scripts. Those characters are not properly supported by the default String API in most mainstream programming languages, and they often require add-on libraries to properly process them. Our experiment results suggest that it is difficult for many programmers to identify instances where the default String API could produce incorrect results in common text manipulation scenarios, despite user education and warnings provided earlier in the experiment in the form of documentation. The findings lead to specific implications for designing programming environments that facilitate the process of making API choices in order to mitigate such misalignments.

7. Acknowledgements

We thank our study participants for their time. In addition, this study would not be possible without the support and feedback from Ian Hickson, Michael Thomsen, Lasse Nielsen and Leaf Petersen from the Flutter team and the Dart team at Google.

8. References

Augier, M. (2000). Rationality, imagination and intelligence: some boundaries in human decision-making. *Industrial and Corporate Change*, *9*(4), 659–681.

Blackwell, A. F., Britton, C., Cox, A., Green, T. R. G., Gurr, C., Kadoda, G., Kutar, M. S., Loomes, M., Nehaniv, C. L., Petre, M., Roast, C., Roe, C., Wong, A., & Young, R. M. (2001). Cognitive

Dimensions of Notations: Design Tools for Cognitive Technology. In *Lecture Notes in Computer Science* (pp. 325–341). https://doi.org/10.1007/3-540-44617-6_31

Clarke, S. (2004). Measuring API usability. *Dr. Dobb's Journal Windows*, S6–S9.

Danesi, M. (2016). *The Semiotics of Emoji: The Rise of Visual Language in the Age of the Internet*. Bloomsbury Publishing.

Davis, M. (2001, March 11). *Text Boundaries (Version 1)*. Unicode Technical Reports. https://www.unicode.org/reports/tr29/tr29-1.html

Davis, M., & Chapman, C. (2020, February 19). *Unicode Text Segmentation (Revision 37)*. Unicode Technical Reports. https://unicode.org/reports/tr29/

Fahl, S., Harbach, M., Perl, H., Koetter, M., & Smith, M. (2013). Rethinking SSL development in an appified world. *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 49–60.

Huh, Y. E., Vosgerau, J., & Morewedge, C. K. (2014). Social defaults: Observed choices become choice defaults. *The Journal of Consumer Research*, *41*(3), 746–760.

Jachimowicz, J. M., Duncan, S., Weber, E. U., & Johnson, E. J. (2019). When and why defaults influence decisions: a meta-analysis of default effects. *Behavioural Public Policy* , *3*(2), 159–186.

McKenzie, C. R. M., Sher, S., Leong, L. M., Müller-Trede, J., & Others. (2018). Constructed preferences, rationality, and choice architecture. *Review of Behavioral Economics*, *5*(3-4), 337–360.

Microsoft. (2018, May 31). *Surrogates and Supplementary Characters*. https://docs.microsoft.com/en-us/windows/win32/intl/surrogates-and-supplementary-characters

Murphy-Hill, E., Sadowski, C., Head, A., Daughtry, J., Macvean, A., Jaspan, C., & Winter, C. (2018). Discovering API Usability Problems at Scale. *2018 IEEE/ACM 2nd International Workshop on API Usage and Evolution (WAPI)*, 14–17.

Myers, B. A., & Stylos, J. (2016). Improving API usability. *Communications of the ACM*, *59*(6), 62–69.

Nielsen, J. (1994, April 24). *10 Heuristics for User Interface Design: Article by Jakob Nielsen*.

Nielsen Norman Group. https://www.nngroup.com/articles/ten-usability-heuristics/

Perkins, D. N., Salomon, G., & Others. (1992). Transfer of learning. *International Encyclopedia of Education*, *2*, 6452–6457.

Piccioni, M., Furia, C. A., & Meyer, B. (2013). An Empirical Study of API Usability. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. https://doi.org/10.1109/esem.2013.14

Schneider, C., Weinmann, M., & Vom Brocke, J. (2018). Digital nudging: guiding online user choices through interface design. *Communications of the ACM*, *61*(7), 67–73.

Shackle, G. L. S. (1964). *General thought-schemes and the economist: the second Woolwich Economic Lecture delivered before the Woolwich Polytechnic on 3 March 1964*. Woolwich Polytechnic, Department of Economics and Business Studies.

Simon, H. A. (1996). *The Sciences of the Artificial - 3rd Edition* (3rd ed.). The MIT Press.

Somers, J. (2017, September 26). The Coming Software Apocalypse. *The Atlantic*. https://www.theatlantic.com/technology/archive/2017/09/saving-the-world-from-code/540393/

Stylos, J., Clarke, S., & Myers, B. A. (2006). Comparing API Design Choices with Usability Studies: A Case Study and Future Directions. *PPIG*, 17.

Stylos, J., & Myers, B. (2007). Mapping the Space of API Design Decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. https://doi.org/10.1109/vlhcc.2007.44

Thaler, R. H., & Sunstein, C. R. (2009). *Nudge: Improving Decisions About Health, Wealth, and Happiness* (Revised & Expanded edition). Penguin Books.

Wikipedia contributors. (2020, August 22). *Languages used on the Internet*. Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Languages_used_on_the_Internet

Cronqvist, H; Thaler, R (2004). Design choices in privatized social security systems: Learning from the Swedish experience. American Economic Review. 94 (2): 424–8.

Dinner, I., Johnson, E. J., Goldstein, D. G., & Liu, K. (2011). Partitioning default effects: why people choose not to choose. Journal of Experimental Psychology: Applied, 17(4), 332.

Jachimowicz, J. M., Duncan, S., Weber, E. U., & Johnson, E. J. (2019). When and why defaults influence decisions: A meta-analysis of default effects. Behavioural Public Policy, 3(2), 159-186.

Johnson, E. J., Shu, S. B., Dellaert, B. G., Fox, C., Goldstein, D. G., Häubl, G., ... & Wansink, B. (2012). Beyond nudges: Tools of a choice architecture. Marketing Letters, 23(2), 487-504.

Johnson, E.J.; Goldstein, D.G. (2003). Do Defaults Save Lives? Science. 302 (5649): 1338–1339.

Kahneman, D., & Tversky, A. (2013). Choices, values, and frames. In Handbook of the fundamentals of financial decision making: Part I (pp. 269-278).

Huh, Y. E., Vosgerau, J., & Morewedge, C. K. (2014). Social defaults: Observed choices become choice defaults. Journal of Consumer Research, 41(3), 746-760.

Larrick, R.P. and Soll, J.B (2008). The MPG Illusion. Science. 320 (5883): 1593–4.

McKenzie, C. R., Sher, S., Leong, L. M., & Müller-Trede, J. (2018). Constructed preferences, rationality, and choice architecture. Review of Behavioral Economics, 5(3-4), 337-360.

Scheibehenne, B., Greifeneder, R. and Todd, P. (2010). Can there ever be too many options? A meta-analytic review of choice overload. Journal of Consumer Research. 37 (3): 409–25.

Schneider, C., Weinmann, M., & Vom Brocke, J. (2018). Digital nudging: guiding online user choices through interface design. Communications of the ACM, 61(7), 67-73.

Appendix: Questionnaire Design

Screener

Thank you for your interest in participating in this research study on the usability of Dart's API. To determine your eligibility, please answer questions on this page.

If you qualify, you can expect to complete the survey within 15 minutes. Please make sure you will not be interrupted. If you opened this survey on a mobile device, we kindly request you to fill it out from a desktop/laptop computer to ensure correct rendering of the content.

By submitting your responses in this survey, you acknowledge that you are at least 18 years of age and that Google and its affiliates may use your responses to improve Google's products and services in accordance with Google Privacy Policy.

#	Question	Criterion
Q2	What is your level of experience with programming in Dart? ● No experience ● Awareness ● Novice ● Intermediate ● Advanced ● Expert	Criterion: must be "Intermediate" or higher to be eligible.
Q3	How proficient are you in English? ● None ● Beginner ● Intermediate ● Advanced ● Native	Criterion: must be "Intermediate" or higher to be eligible.
Q4	How confident are you in using the following Dart APIs? Options ● Async, await and future ● String ● Dates and times ● Math ● Regular expressions Levels ● Not at all confident ● Slightly confident ● Moderately confident ● Quite confident ● Totally confident	Criterion: confidence in using the String API must be "moderately confident" or higher to be eligible.

Introduction

In this survey, we are measuring the experience of reading Dart programs for the purpose of improving its API. You will go through a number of coding scenarios and determine if a given code snippet can satisfy the requirements described in each scenario. Both your answers and the time you spend on answering each question will be recorded in this survey.

Please read the information in the survey carefully, which may help you better assess those coding scenarios. To ensure the validity of the study results, we request you to not share the survey with others. On behalf of the Dart team, thank you for your time.

Scenario 1 (substring)

Note: the purpose of this task is to introduce the grapheme clusters issue to participants in the treatment groups. Treatment group 1 will receive additional conceptual explanation of the problem.

Imagine that you want to implement a function that deletes the last character from a string and returns the result as a new string. The string comes from user input in a text box.

You come across the following snippet online:

```
String skipLastChar(String text) {
  return text.substring(0, text.length - 1);
}
```

Note: the substring (int startIndex, [int endIndex])) method returns the substring of this string that extends from startIndex, inclusive, to endIndex, exclusive. For example, “hello”.substring(0, 4) returns “hell”.

#	Question	Display Logic
Q7	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? ● Yes ● No ● Maybe	
Q9	What do you think could be missing or incorrect in this snippet, if any? ________________	Show if answer to Q7 is “No” or “Maybe”

Scenario 1 feedback

Note: this block will be displayed to both treatment groups but not the control group.

The correct answer is “No”, because the code snippet won’t pass the test below.

Test code:

```
test("skipLastChar(text) removes the last character from the string", () {
    var string = 'Hi 🇩🇰';
    expect(skipLastChar(string), equals('Hi '));
  });
```

Test results:

```
  Expected: 'Hi '
    Actual: 'Hi 🇩???'
     Which: is different. Both strings start the same, but the actual value
also has the following trailing characters: 🇩???
```

It turns out Dart's String API doesn't handle some emojis and non-English characters (e.g., g̈, 가, and षि) well. The formal term for those characters is extended grapheme clusters. The good news is that Dart has an experimental package called characters to handle this kind of situation. Here is how to rewrite this snippet using the package:

```
String skipLastChar(String text) {
  return text.characters.skipLast(1).toString();
}
```

Note that the package provides an extension method which turns a String object into a Characters object.

#	Question	Display Logic & Rationale
Q11	Before participating in this survey, had you run into, or heard about, any issues when using Dart's String API to manipulate text input that includes extended grapheme clusters such as emojis (e.g., 👍 😀 🇩🇰) and accented letters (e.g., café)? ● I had run into such issues before. ● I had heard of such issues but never ran into them. ● I hadn't heard of such issues before. ● I'm not sure.	Rationale: check if the respondents knew about this issue yet still gave the wrong answer.

Additional explanation for treatment group 1

```
String skipLastChar(String text) {
  return text.substring(0, text.length - 1);
}
```

So what was the fundamental problem in this code snippet? There are a couple of things going on here:

- Dart's standard String class uses the UTF-16 encoding, which means that the string is stored in a sequence of 16-bits code units. The length property of String returns the number of those UTF-16 units.
- However, some emojis and non-English characters require more than one UTF-16 code unit to store. For example, 🇩🇰 is stored in 4 UTF-16 code units in a Dart String. Such characters are formally called extended grapheme clusters.

It is clear that when using Dart String's substring method to remove the last "character" from 'Hi 🇩🇰', it will remove the last of the four code units representing the flag, leaving the resulting substring corrupted.

The Characters package was created to address this limitation of the String API. Please take a few minutes to take a look at the documentation of the characters package below:

[Insert screenshot: https://pub.dev/documentation/characters/latest/characters/Characters-class.html]

Transitional page

Next, you will read a few more coding scenarios and evaluate code written to satisfy those scenarios. If you cannot determine whether the code would work or not, please respond "maybe" instead of trying to run the code outside of the survey.

Some scenarios will provide correct answers immediately after you give your assessment, while others will show correct answers at the end of this exercise.

Scenario 2 (email validation)

Note: the purpose of this task is to check if the participants in the two treatment groups would overreact to what they just learned about the String API's limitations in processing grapheme clusters. The snippet in this scenario is actually correct.

You're asked to review code written to implement the following requirements:

- The code is a function that checks if the text input contains a likely email address.
- Since email addresses may include non-English characters these days, the check needs to be inclusive.
- After some testing, the following regular expression is considered good enough: '[^@\.]+@[^@\.]+\.[a-z]*'

```
bool containEmail(String email) {
  return email.contains(RegExp(r'[^@\.]+@[^@\.]+\.[a-z]*'));
}
```

#	Question	Display Logic
Q16	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code and the regular expression have been checked)? ● Yes ● No ● Maybe	
Q18	What do you think could be missing or incorrect in this snippet, if any? ● The code should use the characters package instead. ● The problem is unrelated to the characters package. Please describe the problem briefly: __________ ● I don't know.	Answer to the previous Q is "No" or "Maybe" AND the participant is not in the control group

	Next page	
Q19	The correct answer is "Yes", this snippet can satisfy the requirements described in the scenario. In fact, this scenario requires the String API. First, regular expressions can be used to match text with non-English characters. Second, the characters package doesn't support regular expressions.	Show if the participant is not in the control group.

Scenario 3 (count characters)

Note: the purpose of this task is to check if the participant would realize they need to use the characters package to correctly measure the number of characters in a string.

You're asked to review code written to implement the following requirements:

- The code is a function that checks if the text entered by the user has exceeded a specific number of characters.
- The function returns a positive number of remaining characters if the limit hasn't been reached, or a negative number of extra characters if the limit has been exceeded.

```
int checkMaxLength(String input, int limit) {
  var length = input.length;
  return limit - length;
}
```

#	Question	Display Logic
Q20	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? • Yes • No • Maybe	
Q22	What do you think could be missing or incorrect in this snippet, if any? • The code should use the characters package instead. • The problem is unrelated to the characters package. Please describe the problem briefly: __________ • I don't know.	Answer to the previous Q is "No" or "Maybe" AND not in the control group

Scenario 4 (split string)

Note: the snippet was initially considered to be correct in order to test overreaction, but we later discovered a condition where this snippet would produce incorrect results. Therefore, data from this scenario was excluded from the analysis.

You’re asked to review code written to implement the following requirements:

- The code is a function that splits a string at ‘⭐’ and returns a list of substrings. For example, turning ‘space⭐is⭐for⭐everybody’ into [‘space’, ‘is’, ‘for’, ‘everybody’].

Review the following code snippet and determine if it can correctly meet your requirements.

```
List splitStarSeparatedWords(String text) {
  return text.split('⭐');
}
```

#	Question	Display Logic
Q23	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? ● Yes ● No ● Maybe	
Q25	What do you think could be missing or incorrect in this snippet, if any? ● The code should use the characters package instead. ● The problem is unrelated to the characters package. Please describe the problem briefly: __________ ● I don’t know.	Answer to the previous Q is “No” or “Maybe” AND not in the control group
	Next page	
Q26	The correct answer is “Yes”, this snippet can satisfy the requirements described in the scenario. The snippet works because the String API’s split method can appropriately handle emojis as separators. Moreover, the characters package doesn’t have a split method or its equivalent.	Show if the participant is not in the control group.

Scenario 5 (create initials)

Note: the purpose of this task is to test if the participant would realize that the index operator can break grapheme clusters.

You’re asked to review code written to implement the following requirements:

- The code is a function that creates initials from the first name and the last name the user enters in two separate text fields.
- For example, the function needs to generate initials such as “JS” from “john” and “smith”.

```
String createInitials(String firstName, String lastName) {
    return firstName[0].toUpperCase() + lastName[0].toUpperCase();
  }
```

#	Question	Display Logic
Q27	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? ● Yes ● No ● Maybe	
Q29	What do you think could be missing or incorrect in this snippet, if any? ● The code should use the characters package instead. ● The problem is unrelated to the characters package. Please describe the problem briefly: _________ ● I don't know.	Answer to the previous Q is "No" or "Maybe" AND not in the control group

Scenario 6 (text overflow ellipsis)

Note: the purpose of this scenario is to check if the participant still remembers the substring scenario where they were first exposed to the grapheme manipulation issue.

Your app needs to display a list of messages, one per line. You're asked to review code written to implement the following requirements:

- The code is a function that displays text overflow as an ellipsis when the message's length exceeds the given character limit.
- For the purpose of this survey, you can ignore the varying widths of characters.

```
String textOverflowEllipsis(String text, int limit) {
  if (text.length > limit) {
    return text.substring(0, limit - 3) + '...';
  } else {
    return text;
  }
}
```

	Question	Display Logic
Q30	Does this snippet correctly satisfy the requirements described in the scenario (assuming the syntax of the code has been checked)? ● Yes ● No ● Maybe	
Q32	What do you think could be missing or incorrect in this snippet, if any? ● The code should use the characters package instead. ● The problem is unrelated to the characters package. Please describe the problem briefly: _________ ● I don't know.	Answer to the previous Q is "No" or "Maybe" AND not in the control group

Review correct answers

Please review the expected answers to the questions in the previous code scenarios. This page also shows the correct code for coding scenarios where the snippets provided had issues.

Summary of correct answers

The table below shows the expected responses to the question "Will the code above correctly satisfy the requirements described in the scenario?"

Scenario #	Your Answer	Correct Answer
2 (email validation)	<Insert the participant's answer>	Yes
3 (count characters)	<Insert the participant's answer>	No
4 (split string)	<Insert the participant's answer>	Yes
5 (create initials)	<Insert the participant's answer>	No
6 (text overflow ellipsis)	<Insert the participant's answer>	No

Scenario 2 (email validation)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: Yes.
- This scenario requires the String API. First, regular expressions can be used to match text with grapheme clusters. Second, the characters package doesn't support regular expressions.

Scenario 3 (count characters)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No. The code won't pass the test below.
- Test code:

```
test("checkMaxLength(String input, int limit) returns how many characters left
in the space", (){
  var input = "Laughter is the sensation of feeling good all over and showing
it principally in one place.";
  var limit = 140;
  expect(checkMaxLength(input, limit), equals(49));
  input = "Laughter 😀 is the sensation of feeling good all over and showing
it principally in one place.";
  expect(checkMaxLength(input, limit), equals(47));
});
```

- Test results:

```
00:00 +1 -2: checkMaxLength(String input, int limit) returns how many
characters left in the space [E]
  Expected: <47>
    Actual: <46>

  package:test_api  expect
  test.dart 22:5    main.<fn>
```

- Correct code:

```
int checkMaxLength(String input, int limit) {
  var length = input.characters.length;
  return limit - length;
}
```

Scenario 4 (split string)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: Yes
- The snippet works because the String API's split method can appropriately handle emojis as separators. Moreover, the Characters class doesn't have a split method or its equivalent.

Scenario 5 (create initials)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No. The reason is that the subscript interface as used in name[0] could grab a fraction of a character. For example, the code won't pass the test below, because the character "É" can be a combination of an "E" and an acute "´".
- Test code:

```
test(
      "createInitials(firstName, lastname) creates initials from a first name
and a last name",
      () {
    var firstName = "étienne";
    var lastname = "bézout";
    expect(td.createInitials(firstName, lastname), equals('ÉB'));
});
```

- Test result:

```
00:01 +1 -5: createInitials(firstName, lastname) creates initials from a first
name and a last name [E]
  Expected: 'ÉB'
    Actual: 'EB'
     Which: is different.
            Expected: ÉB
              Actual: EB
                       ^
             Differ at offset 1
```

- Correct code:

```
String createInitials(String firstName, String lastName) {
  var initials = firstName.characters.first.toUpperCase() +
      lastName.characters.first.toUpperCase();
  return initials;
}
```

Scenario 6 (text overflow ellipsis)

Will the code above correctly satisfy the requirements described in the scenario?

- Correct answer: No, the code won't pass the test below.

- Test code:

```
test(
    "textOverflowEllipsis(String text, int limit) displays an ellipsis for
overflown text",
    () {
  var input = "🦏rhinoceros";
  var limit = 7;
  expect(td.textOverflowEllipsis(input, limit), equals("🦏rhi..."));
});
```

- Test result:

```
00:01 +1 -3: textOverflowEllipsis(String text, int limit) displays an ellipsis
for overflown text [E]
  Expected: '🦏rhi...'
    Actual: '🦏rh...'
     Which: is different.
            Expected: 🦏rhi...
              Actual: 🦏rh...
                          ^
             Differ at offset 4
```

- Correct code:

```
String textOverflowEllipsis(String text, int limit) {
  return text.characters.take(limit - 3).toString() + '...';
}
```

Attitudes and preferences

(Display logic: treatment groups only)

#	Question	Display Logic & Rationale
Q37	How important is it to handle emojis correctly in text manipulation tasks while developing apps? ● Extremely important ● Very important ● Moderately Important ● Slightly important ● Not at all important	Rationale: To measure perceived importance of the issue.
Q38	Which of the following statements best describes how you would approach text manipulation in Dart in the future? ● I would stick with the String API unless I have evidence that something is broken for my app's users. ● I would evaluate every use case of the String API in my project and decide when the character package should be used instead.	Rationale: To measure potential behavior change

	● I would use the characters package to manipulate text whenever it can be used. ● I'm not sure. ● Other: ____________	
Q39	How easy is it to determine when you need to use the characters package to write correct code. ● Very easy ● Somewhat easy ● Neutral ● Somewhat hard ● Very hard	Rationale: To measure confidence in making intuitive choices.
Q40	In your own words, how would you summarize situations where the characters package should be used to properly manipulate text? ___________	Rationale: To measure, how well the user can come up with rules of thumb they can remember. A rubric needs to be created to grade this answer.
Q41	If Dart's String API can provide different views into the data stored in a String object, which do you prefer to be the default view: ● UTF-16. This is the current behavior. Random access by string index (e.g., input[2]) is efficient, but it could break some emojis and accented letters. ● Grapheme Clusters. All emojis will be handled correctly out of the box. Random access by string index (e.g., input[2]) can be simulated but it will be more expensive. ● I'm not sure.	Rationale: To measure user preference of default string view.
Q42	Why do you prefer the option you chose in the previous question? _________________	

Participant background

#	Question	Display Logic & Rationale
	You're almost done. We have just a couple more questions on this page.	
Q43	Which types of apps have you developed using Dart/Flutter? (Select all that apply) ● Business and productivity tools (e.g., calendar, to-do) ● Communication and social network ● Education ● Enterprise ● Financing and banking ● Health ● Games	Rationale: Apps in categories such as "business and productivity tools" or "communication and social network" are more likely to have user input that includes EGC.

	• Lifestyle, food, and drink (e.g. shopping, news, travel, fitness) • Music and Video • Utilities (e.g., weather, calculator) • Sports • Other	
Q44	Is English the primary language in your country? • Yes • No	Rationale: Developers in non-English speaking countries might be more aware of the EGC issue.
Q45	Before participating in this survey, had you run into, or heard about, any issues when using Dart's String API to manipulate text input that includes extended grapheme clusters such as emojis (e.g., 👍 😀 🇩🇰) and accented letters (e.g., café)? • I had run into such issues before. • I had heard of such issues but never ran into them. • I hadn't heard of such issues before. • I'm not sure.	Logic: show if condition = control. Respondents in treatment conditions get this question right after scenario 1. Rationale: check if the respondents knew about this issue yet still gave the wrong answer.
Q46	Which of the following statements best describes your knowledge about the characters package for Dart before participating in this study? • I didn't know this package existed before the study. • I heard about it but knew little about what it does. • I knew the package and the problem it solves, but I haven't used it. • I've used the characters package in my own projects.	
Q47	What is your level of experience with programming in Swift? • No experience • Awareness • Novice • Intermediate • Advanced • Expert	
Q48	How often did you have trouble understanding something in this survey (e.g., questions, instructions, or descriptions)? • Always • Often • Sometimes • Rarely • Never	
Q49	What was unclear in this survey? __________	Logic: show if the answer to Q48 is not "Rarely" or "Never".