Universiteit Leiden
The Netherlands

Opleiding Informatica & Economie

EVALUATING A NEW JAVASCRIPT FEATURE

for a Thesis

Kousar Sedigi

Supervisors:
Felienne Hermans & Joost Visser

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

23/08/2023

Abstract

The paper presents qualitative research results on the Do-Expression, a proposed feature in JavaScript.The study uses interviews with the think-aloud method to gather data on using the Do-expression in JavaScript. Three expert programmers with over five years of experience and three participants with 0 to 5 years experience in JavaScript were interviewed online in June 2021. The interview was divided into four parts to understand participants' initial reactions and perceptions of the Do-expression and their ability to distinguish it from the Do-while statement and search for resources. The data collected was used for the quantitative analysis of the study.

The paper highlights the importance of careful consideration and proper documentation for introducing the Do-expression to avoid confusion and ensure its proper usage. The experts' attention to detail and experience with programming influenced their interpretation of the feature, while beginners' lack of familiarity with expression-oriented programming affected their understanding of it. It was found that experts consider the Do-expression syntax in JavaScript a helpful feature and believe it should be developed further. In contrast, beginners were initially confused but eventually understood its purpose. However, they needed to be convinced of its usefulness and were unlikely to use it in their work. The insights from the interviews will inform the development of a future survey with clear and precise questions and examples to gather more comprehensive data on programmers' perspectives on the Do-expression syntax.

Contents

1 Introduction

The first step in adding a new feature to a programming language is to identify a need or desire for the feature. This requirement could come from various sources, including users of the language, developers who work on the language, or simply from observation of trends in other programming languages.

Once a need has been identified, the next step is typically to propose the new feature to the community that develops and maintains the language. Proposing a new feature could involve:

- submitting a formal proposal.
- discussing the idea on a mailing list or forum.
- presenting the idea at a conference or gathering of language users and developers.

The community will then evaluate the proposed feature to determine whether it is desirable, feasible, and appropriate for the language. This evaluation may involve a variety of criteria, including the potential impact on existing code, the clarity and consistency of the proposed syntax or semantics, and the compatibility of the feature with the broader goals and design principles of the language.

If the community decides that the new feature is a good fit for the language, the next step is to formalize the feature. This process could involve writing a specification that defines the syntax and semantics of the feature, or it could involve writing a prototype implementation to test the feature in practice.

Before the new feature can be added to the language, it may need to undergo a testing and review process to ensure that it is stable and well-designed. This process could involve creating test cases to verify the correctness and performance of the feature, soliciting feedback from users and developers, or undergoing a formal review by language experts or standards bodies.

Once the new feature has been approved and formalized, it can be added to the language in a future release. However, adding new features to a programming language can be challenging, as it requires balancing the needs of existing users with the desire to innovate and improve the language. In some cases, new features can introduce new bugs or make it more difficult to write correct code, so it is essential to consider the costs and benefits of each proposed feature before adding it to the language.

In this paper, we are going to look at do-expression. We will look at the benefits and the drawbacks by doing experiments. Based on the experiments, we will decide if we will add this feature to JavaScript?

```
1 //Do-Expression
2 let value = do {
3             let temp = x + 1;
4             temp * temp
5             }
```

Listing 1: Example of Do-Expression

1.0.1 Research Question

To understand how confusing Do-expression is, the main question we want to answer in this thesis is:

Would JavaScript users benefit from adding Do-Expression to the programming language?

We will do that by answering the following sub-questions:

- What are the different interpretations of the new Do-Expression by JavaScript programmers?
- Will the programmers have a better understanding after priming?
- Will programmers confuse Do-Expression with the do-while statement?

1.1 Thesis overview

The paper will likely begin by introducing the Do-Expression feature and explaining its syntax and semantics. Section 3 will also provide some background on the motivation for adding this feature to JavaScript, such as the benefits of local block scoping and the potential for reducing memory usage.

The paper will then describe the experiments that were conducted to evaluate the Do-Expression feature in section 4. This could involve creating test cases that exercise the feature in various contexts or analyzing existing code to determine how the feature could be used in practice. The experiments may also compare the performance of code that uses DO-Expressions to equivalent code that does not use the feature.

The outcome will be discussed in Section 5. Based on the results of the experiments, the paper will then evaluate the benefits and drawbacks of the DO-Expression feature for potential inclusion in JavaScript. It may consider factors such as ease of use, performance impact, and compatibility with existing code and programming practices.

Finally, In Section 6 a recommendation will be made on whether or not to add the Do-Expression feature to JavaScript. This could involve proposing modifications to the feature based on the experimental results, or simply recommending that the feature be included as-is. The paper may also identify areas for further research or development related to the DO-Expression feature or related features in JavaScript. Section 7 we will interpret and describe the significance of our findings.

This bachelor thesis was part of the PERL group at LIACS, made possible by the support of Felienne Hermans and Yulia Startsev.

2 Background

JavaScript is one of the most widely used programming languages in the world today, and it has a rich history that dates back to the mid-1990s. The very first version of JavaScript was developed by Brendan Eich in just 10 days in May 1995. At the time, the language was called Mocha.

A year after Mocha was first developed, its name was changed to LiveScript. This was done to reflect the fact that the language was designed to be executed in real-time, as users interacted with web pages. However, in an effort to capitalize on the popularity of the Java programming language, LiveScript was renamed JavaScript in December 1995.

It's important to note that despite its name, JavaScript has nothing to do with the Java programming language. JavaScript is a completely separate programming language, and its design and syntax are quite different from Java. The decision to name the language JavaScript was largely a marketing move, aimed at attracting Java developers to the new language. Despite this, JavaScript has gone on to become one of the most widely used programming languages in the world, and its popularity continues to grow with each passing year.

To standardize the language , ECMA releases ECMAScript 1 (ES1) in 1997. The first standard for JavaScript. In 2015 ECMA releases the biggest update ever the ES5(ECMAScript 5) and after that ECMA changes to annual releases in order to ship less features per update.

In order to ensure consistency across different implementations of JavaScript, the European Computer Manufacturers Association (ECMA) developed a standardized version of the language known as ECMAScript. ECMAScript defines the syntax and semantics of the JavaScript language.

The first version of ECMAScript, known as ES1, was released in 1997. This was the first standardized version of the JavaScript language. ECMAScript 1 was designed to be compatible with the features of the original version of JavaScript, while also adding some new features such as regular expressions and exception handling.

Over time, new versions of ECMAScript were released to add new features and improve the language. ECMAScript 3 (ES3) was released in 1999, and it introduced several new features such as try-catch statements, named function expressions, and more flexible regular expressions.

One of the biggest updates to the language came with the release of ECMAScript 5 (ES5) in 2009. ES5 added several new features such as strict mode, which introduced a subset of the language with stronger error checking and more restrictive rules.

Ecma TC39 is a group of JavaScript developers, implementers and more. They maintain and evolve the definition of Javascript. TC39 meets every two months to discuss the proposals. There are 5 stages and Do-expression is in stage 1[TC3].

TC39 meets every two months to discuss proposals for new features and changes to the language. The proposals go through a rigorous process of review, discussion, and testing before they are

accepted into the standard. The goal of this process is to ensure that new features are well-designed, thoroughly tested, and broadly useful to the JavaScript community.

To help manage the proposals, TC39 has developed a process that divides proposals into five different stages. Each stage represents a different level of maturity for a proposal, with stage 0 being the least mature and stage 4 being the most mature.

Do-expressions is a proposal that is currently in Stage 1 of the process. This means that it is still in the early stages of development, and there is no guarantee that it will be included in the final standard. However, the fact that it has progressed to Stage 1 is a good sign that the TC39 committee sees potential in the proposal, and it will continue to be discussed and refined in future meetings.

In JavaScript, expressions and statements are two distinct categories of code constructs. An expression is a piece of code that produces a value, while a statement is a complete instruction that performs an action.
For example, 12 + (7 +2) is an expression that evaluates to the value 3, while console.log("Hello, world!") is a statement that prints a message to the console.

```
//Expressions return some values.
Example -> 12 + (7 + 2)
//return value is 21

//Statements just perform some actions but do not produce any value.
console.log("Hello, world!")
            }
```

Listing 2: Example of Expression and statement

Do-expressions blur the line between expressions and statements by allowing you to put statements inside an expression. This means that you can write a block of code that produces a value, without having to use a separate function or variable assignment.
The next example will show us one of the benefits of Do-expression.

```
//Not using do Expression
let temp = x + 1;
let value = temp * temp;

//With do Expression
let value = do {
            let temp = x + 1;
            temp * temp
            }
```

Listing 3: Example of code with and without do-expression

As it is shown in the example. The semantics is possible , only the syntax is changed. Now temp has become a local variable and after the block the value of temp is null.
There are also confusions about do Expression. For example , not being able to figure out what the expected outcome would be.

The example given in the listing 3 suggests that a do-expression can be used to create a block of code that produces a value, similar to a function, but with a more concise syntax. Specifically, the example shows that a temp variable can be created and assigned a value within a do-expression block, and this variable is local to the block, which means that it is not accessible outside the block.

As for the confusion around do-expression, this is to be expected with any new language feature. When a new feature is proposed, it needs to be thoroughly tested and evaluated to determine its usefulness, safety, and compatibility with other language features. This process can take some time and may involve several rounds of revisions.

Additionally, since do-expressions blur the line between expressions and statements, it may take some time for developers to get used to the new syntax and understand how it works in different contexts. It is possible that some confusion or ambiguity will arise as developers start using do-expressions in real-world code.

Overall, while do-expressions offer a new way to write expressive code in JavaScript, it is important to carefully consider their benefits and drawbacks, and to thoroughly test and evaluate their behavior in various contexts, before deciding whether to adopt them in your own code.

Do-expression make use of completion values. Completion value is the value that is returned when a statement completes. Sometimes this contains a value but not always. You might be familiar with this when you are assigning an expression.
In JavaScript, a completion value is the result of evaluating a statement or expression. A completion value can contain a value, or it can indicate that the statement or expression completed abruptly, for example, by throwing an error.

Do-expressions make use of completion values by allowing you to specify a statement block as the expression. When the statement block completes, the completion value of the last statement in the block is returned as the value of the do-expression.

For example, consider the following do-expression:

```
let x = do {
  if (someCondition) {
    "foo";
  } else {
    "bar";
  }
};
```

Listing 4: Do-Expression

In this example, the do-expression contains an if-else statement block. If someCondition is true, the string "foo" is returned as the completion value of the block. Otherwise, the string "bar" is returned. The completion value of the do-expression is the same as the completion value of the block, so in this case, the value of x is either "foo" or "bar".

Completion values can be tricky to work with, as they can contain values or exceptions, and their

behavior can depend on the context in which they are used. It is important to understand how completion values work in JavaScript and to use them carefully to avoid unexpected behavior in your code.

```
//undefined completion value
let foo = 1;

//defined completion value
1;
```

Listing 5: Completion values

Completion records in JavaScript are objects that include a type and a value. The type of a completion record indicates why a statement or expression completed, and can be one of the following:

- normal: the statement or expression completed normally, without any unusual behavior.
- return: the statement or expression completed with a return statement.
- throw: the statement or expression completed with an exception being thrown.
- break: the statement completed with a break statement.
- continue: the statement completed with a continue statement.

In each of these cases, the completion record may also include a value, which is the value that is returned or thrown by the statement or expression.

Do expressions make use of normal completions, which means that the completion value of a do expression is the value of the last statement in the block, unless that statement completes with an exception or a return statement. If the last statement completes normally, its value is used as the completion value of the do expression.

Due to some potential for confusion there are some limitations. You can't use Do-expression with a declaration, an if without an else statement or a loop.

```
//Do-expression with a declaration
(do {
  let x = 1;
});
//Do-expression with an if without else
(do {
  if (foo) {
    bar
  }
});
//Do-expression with loop
(do {
  while (cond) {
    // do something
```

```
15   }
16 });
```

Listing 6: Limitation of do Expression

3 Related Work

The decision-making process for adding new features to JavaScript is not scientific, as it involves a discussion among members of TC39. While the champion presents their proposal, other members of the committee can provide their input and opinions on the proposal, leading to a discussion. Ultimately, the decision to accept or reject a proposal is made by the consensus of the committee members. This approach can lead to both positive and negative outcomes. On the one hand, proposals with potential issues or problems can still be approved if they have a strong advocate or significant supporters. On the other hand, proposals that could bring significant benefits to the language may only be accepted if they are well-presented or if there is enough interest or support among the committee members. The decision-making process can also be affected by other factors, such as the technical feasibility of the proposal, its impact on existing code, and its potential adoption by developers. Therefore, it is essential for proposal champions to be well-prepared and make a strong case for their proposals while also being open to feedback and criticism from other members of the committee.

Although not much research has been done on evaluating a single feature. There are papers on evaluating a programming language. Steven Clarke used the Cognitive Dimensions framework to evaluate a new programming language[Cla01]. The Cognitive Dimensions framework is a framework for evaluating the usability of programming languages based on different cognitive dimensions such as viscosity, hidden dependencies, and error proneness. It is often used to compare different programming languages or language features, as it provides a structured way to evaluate these dimensions' impact on a language's usability.Steven Clarke's paper this framework to evaluate a new programming language called Mondrian. The paper compares the effectiveness of two evaluation methods: a lab-based evaluation and a questionnaire-based evaluation. The lab-based evaluation involved a group of participants performing tasks in the Mondrian language while being observed and interviewed by researchers. The questionnaire-based evaluation involved a larger group of participants filling out a survey about their experience with the language. The paper concludes that both evaluation methods can be helpful, but the questionnaire-based approach is more practical for more extensive scale evaluations. However, the paper also notes that the questionnaire format needs to be improved to analyze a language or language feature comprehensively. Overall, while there may not be a lot of research specifically on evaluating single language features like the Do-expression in JavaScript, frameworks like the Cognitive Dimensions can be applied to provide structured and comprehensive evaluations of programming languages and their features.

Alan F. Blackwell and Thomas R.G. Green proposed a generalized questionnaire based on the Cognitive Dimensions of Notations framework for evaluating programming languages[BG00]. The Cognitive Dimensions framework provides a systematic and comprehensive set of criteria to evaluate the usability of a programming language. However, the authors noted that the use of language in the questionnaire is crucial to ensure that respondents clearly understand what is required of them.

The authors suggest that respondents should be able to choose which features of the language they want to evaluate, and the questionnaire should be designed to be flexible and customizable to suit different languages and contexts. They also emphasize the importance of clear instructions and explanations of the questions so that respondents can provide accurate and meaningful feedback. The authors also suggest that the questionnaire approach can be supplemented with other evaluation methods, such as user testing and expert reviews. A more comprehensive and accurate assessment of a programming language can be obtained by combining multiple evaluation methods.

Keertipati, S., Licorish, S. A., and Savarimuthu explored the decision-making process in Python[KLS16]. The focus was on the normative decision-making process within Open Source Software (OSS) projects, specifically Python. Their study uses Python Enhancement Proposals (PEPs) to explore aspects of the normative decision-making processes in OSS development. The research compared the extracted process models and the processes promoted by the Python community. The aim was to assess the level of overlap between these two sets of processes. Through interviews, surveys, and analysis of community discussions and documentation, the researchers examine the decision-making dynamics in Python. They explore how decisions are made, the criteria used for feature selection, and the roles of stakeholders, including core developers, community contributors, and the Python Steering Council. The findings revealed significant differences between the extracted and officially advertised processes by the Python community. The extracted processes were found to be more complex. Furthermore, the success of PEPs was often attributed to the significant contributions of key members within the community. Overall, this paper contributes to a deeper understanding of the decision-making processes in Python and offers insights into the mechanisms that shape the language's evolution. It provides valuable information for language designers, developers, and stakeholders interested in the decision-making dynamics of programming languages.

Crowston, K. and Howison, J. research examine communication patterns within Free/Libre Open Source Software (FLOSS) development teams [CH06]. This study aims to understand the social structure of these teams and its impact on collaboration and team performance. Using social network analysis, the researchers analyze interactions in 62,110 bug reports from 122 large and active projects. Contrary to expectations, the study's findings shed light on the diverse communication structures observed within FLOSS teams. Interestingly, some groups demonstrate high centralization, while others deviate from this pattern. The research highlights that FLOSS projects are mostly hierarchical, which aligns with past research but challenges the popular image of these projects as non-hierarchical. By studying communication patterns and social structures, this research sheds light on the organizational dynamics of FLOSS projects. It emphasizes the importance of communication structures in facilitating effective collaboration and team performance in open-source development. Overall, these findings contribute to understanding distributed work in FLOSS development teams.

Similar findings were found in the study done by Crowston, K. Wei, K. Howison J, and Wiggins, A [CWHW08]. Their study offers a comprehensive analysis of the existing knowledge of Free/Libre Open Source Software (FLOSS) development. The paper conducts a systematic review of the literature, explaining key insights while identifying gaps that remain unexplored. The authors emphasize the distinctive attributes of FLOSS development teams as an ideal context for studying distributed work. They explore the complex social structures persisting within these teams, giving insights into

the dynamics of social networks and the presence (or absence) of hierarchical arrangements in FLOSS projects. The findings challenge common beliefs, revealing instances of both highly centralized and non-traditional hierarchical structures across teams. Moreover, the article investigates the relationship between project size and centralization, unearthing a negative correlation that suggests larger projects tend to adopt more modular organizational structures. Overall, this scholarly work provides valuable insights into the current state of knowledge pertaining to FLOSS development. By outlining the knowns and unknowns, this research paves the way for future investigations in this domain.

Sharma, P. N., Savarimuthu, B. T. R., and Stanger, N. focused on the extraction of decision-making rationale from the Python email archives in their study [SSS21]. The study aims to understand the decision-making processes and the underlying rationale behind them in the context of open-source software development. Uncovering the underlying rationale behind these decisions encourages transparency by bringing them to light. The authors use natural language processing techniques to analyze a large collection of email messages exchanged among Python developers. Through their analysis, they identify and extract key pieces of information related to decision-making, such as reasons, justifications, and discussions. The findings of the study provide valuable insights into the decision-making practices within the Python community and shed light on the factors affecting the development of open-source software projects. The research contributes to the understanding of how decisions are made and documented in the context of open-source software development.

4 Approach

In this paper discussing Do-expression, a mixed-method approach is used to evaluate the feature. Mixed methods research involves combining both qualitative and quantitative research methods to provide a complete understanding of the research question at hand.[Lit18].
In the case of evaluating the Do-expression feature in JavaScript, the qualitative research method is chosen to gain an in-depth understanding of the issues and challenges that programmers may face when encountering this feature. This process involves interviews with programmers about the feature. Qualitative research can provide a detailed understanding of the attitudes and perspectives of users toward the feature.

After using qualitative methods, the authors will also use a quantitative approach to gather data from a broader audience. This may involve conducting surveys or experiments to measure the impact of the Do-expression feature on programmers' productivity, code quality, or other metrics. Quantitative research can provide more objective and generalizable results and help confirm or refute qualitative research findings.

Overall, the mixed method approach allows the authors to combine the strengths of both qualitative and quantitative research methods and provides a complete understanding of the impact of the Do-expression feature on JavaScript programming.

4.1 Qualitative method

For our quantitative method, we have chosen interviews. During the interviews, we will use the think-aloud method. The think-aloud method is a technique that requires participants to verbalize their thoughts as they work through a task.This approach is used to correctly identify the issues a programmer might face [VSBS94].

The interviews were conducted online since the participants were located in different places. This approach is helpful as it allows for a more flexible and convenient way to collect data. Online interviews also allow a larger pool of participants to be reached, making it possible to get a more diverse set of responses.

During the interviews, participants will be asked to provide feedback on their experience using the Do-expression, including any challenges they faced and any suggestions for improvement. This data will be collected and analyzed to help answer the research question and identify potential Do-expression issues.

Overall, the think-aloud method is a helpful tool for understanding how participants approach a task and can provide valuable insights into the thought processes involved in using a new feature such as the Do-expression.

4.1.1 Participants

In order to have a diverse group of participants and to capture a broad range of perspectives, it is essential to choose participants with different levels of experience. In this study, we interviewed three expert programmers with more than five years of experience in JavaScript and three participants with between 0 and 5 years of experience with JavaScript.

To recruit these participants, we contacted our professional network and used various online platforms to find potential participants who met their criteria. Having a diverse group of participants is essential because it can help uncover a broader range of issues and perspectives related to the topic of study.
The interviews were conducted in June 2021 and likely followed a structured protocol to ensure consistency in the data collected. The think-aloud method was used during the interviews to encourage participants to verbalize their thought processes as they worked through tasks related to the Do-expression in JavaScript. This approach can provide rich data on programmers' difficulties and challenges when using this new feature.

4.1.2 Procedure

The interview was designed to explore the participants' understanding and perceptions of the Do-expression in JavaScript. The first part of the interview was focused on getting to know the participant and their level of experience with JavaScript. This part helped the interviewer to understand the context and background of the participant.

In the second part of the interview, the participants were shown some examples of Do-expression and were asked what they thought the Do-expression does when encountered for the first time. This part allowed the interviewer to understand the initial reaction and perception of the participants toward the new feature.

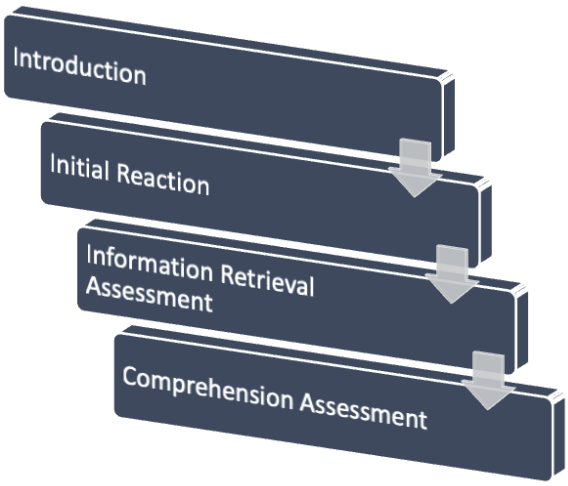

Figure 1: Procedure

In the third part of the interview, the participants were asked if they had ever heard of the Do-while statement in JavaScript and if they could distinguish between the two. This part was essential to identify any confusion that might arise due to the similarity of names.The participants were also asked how they would search for more information about Do-expression if they encountered it for the first time. This assessed their ability to search for resources and learn new language features.The participants were also asked how they would search for more information about Do-expression if they encountered it for the first time. This was to assess their ability to search for resources and learn new features of the language.

In the final part of the interview, the same examples from the second part were shown again, but the participants explained the Do-expression and how it works. This part evaluated the explanation's effectiveness and the participant's ability to understand the feature afterward.

5 Results

The qualitative research results suggest some need for clarification among programmers when they encounter the Do-expression for the first time. The participants were unfamiliar with the concept of "expression-oriented programming" and, thus, found it difficult to understand the semantics of the Do-expression. Some participants also confused it with the existing Do-while statement, which caused further confusion.
However, after receiving an explanation and understanding the concept of expression-oriented programming, the participants found the Do-expression valuable and easy to understand. They appreciated its flexibility and could see its potential for improving code readability and reducing redundancy.
Overall, the qualitative research suggests that the Do-expression has the potential to be a helpful feature in JavaScript. However, its introduction may require careful consideration and proper documentation to avoid confusion and ensure proper usage.

5.1 Qualitative method

We interviewed 6 participants, of which three were experts and three with experience of fewer than five years. Although we have used the same examples, the findings were different. First, we will go through the interviews with the experts, and then we will discuss the results from the beginners.
During the interviews with the experts, it was observed that they could understand the concept of Do-expression quite easily. They identified that the Do-expression is a new construct that has been added to the language and is different from the Do-while statement. However, they did mention that the syntax might take some getting used to as it differs from what they are used to.
When presented with examples of the Do-expression in use, the experts could quickly identify what the code was doing and how the Do-expression was being used. They also mentioned that the Do-expression could be helpful in specific scenarios, especially when dealing with asynchronous code.

However, despite their familiarity with the language and experience, the experts mentioned some concerns about Do-expression. For example, they mentioned that the syntax of the Do-expression might need to be clarified for beginners and that it might take some time for developers to get used to it. They also mentioned that there might be some compatibility issues with older browsers and that this might make it challenging to use the Do-expression in production code.

Overall, the experts were generally positive about the Do-expression and felt that it could be a valuable addition to the language, provided that developers take the time to understand how it works and can be used effectively.

5.1.1 Experts

The experts only knew that we would discuss a new JavaScript feature. All three of the participants have around ten years of experience.
The participants were comfortable discussing their thoughts and opinions with the interviewer, which could be due to their level of expertise and confidence in their knowledge. Additionally, since the participants had no prior knowledge of the specific feature being discussed, they may have approached it with curiosity and openness. This could have contributed to their engaged and detailed answers.

In the second part, where we showed the examples to the experts, we noticed that they were looking at the whole code rather than only focusing on the Do-expression part. As is shown in the following example, in line 4, a semicolon was not used, but in line 9, we did. Do-expression has nothing to do with a semicolon; the experts still noticed it was missing.

The behavior observed during the interview with experts is expected. Experienced programmers are trained to look for subtle details in code and often scan the entire code rather than just focusing on the specific element under consideration. The fact that the experts noticed the missing semicolon is a testament to their attention to detail and experience with JavaScript.

It is worth noting that the missing semicolon is unrelated to the Do-expression itself and would not affect the code's behavior significantly. The experts may have noticed it simply because it goes against the usual style guidelines for writing JavaScript code.

```
//first example
let x = do {
    let tmp = 5;
    tmp * tmp + 1
  };

//second example
  let x = do {
    let y = 1;
  };
```

Listing 7: With and without semicolon

Also in the following example we were trying to find out what participants think happens when we use the try catch statement. Every expert thought it would return null because the blob statement was wrong. In line 3 *Jan* and the number *42* also need quotation marks around it. Therefore experts taught the funtion would return null rather than parse it.
This example demonstrates how the experts' experience with programming and syntax rules influenced their interpretation of the Do-expression. They were able to quickly identify syntax errors and the incorrect use of variables without fully understanding the purpose of the Do-expression. It highlights how prior knowledge and experience can impact the interpretation and understanding of new language features. This is an important consideration when evaluating the adoption and usage of new language features, as experienced developers may have different expectations and use cases than beginners.

```
//(Conditionals // control flow)

const blob = '{"userid":Jan, "age":42}';

function getUserId(blob) {
let obj = do {
  try {
    JSON.parse(blob)
  } catch {
    return null;
  }
};
return (obj.userId);
}
//correct output : Jan
```

Listing 8: Try/catch statement

In part 3 none of the experts confused the Do-expression with the Do-while statement. Also Do-while is not used by them in their daily life. Explaining what Do-expression is about went smoothly since all of them had already a good understanding of what a completion value is. So we only needed to explain what a completion value is in the context of a do expression. Also all of them would use MDN which is a web page for resources for developers, by developers in order to find more about the expression.
Elaborating on the use of MDN, it is a widely used resource for developers to find information about programming languages, APIs, and other technical resources. The fact that all experts mentioned it shows that it is a reliable source of information for them. This also highlights the importance of

having good documentation for new programming features like the Do-expression, as developers often rely on these resources to learn and understand new concepts.

In the fourth section of our study, we revisited the examples and provided a comprehensive explanation of the Do-expression. Interestingly, the majority of the participants responses remained consistent with their previous answers. In Example 5, presented in Listing 9, participants accurately predicted the expected outcome both before and after the explanation. This suggests that even without prior clarification, participants possessed a sufficient understanding of the Do-expression and its potential outputs. Example 9, also presented in Listing 9., During the exploration of conditionals. With the while loop, some participants expressed uncertainties. They raised concerns about what would happen if the loop did not execute. After providing clarification that the while loop has certain limitations and not all scenarios are permitted, their doubts were addressed. After that, no other confusion arose, and the participants expressed an interest in understanding the specific limitations associated with the while loop.

```
1
2  //Example 5:
3  let age = do {
4      let myMomsAge = currentYear() - 1960;
5      myMomsAge + 1;
6  }
7
8  //Example 9:
9
10 let AmountOfShoes = do {
11     while (ShoeRack != 0) {
12     ShoeRack--;
13     let x = x + 1;
14     }
15 }
```

Listing 9: Example 5 9 from the interview

Overall they could see how this expression could be useful and wanted to know more about it.

5.1.2 Beginners

The participants mentioned they had almost no experience with JavaScript and were working with other programming languages more often. One of the participants just started working and the two others are in the last year of their bachelor. The experience they had was between 2 to 6 months.

During the second segment of the interview, when the participants were presented with examples featuring the Do-expression, they demonstrated confusion because of their limited experience in JavaScript. They needed help comprehending the complexities of this functionality and its syntax. They also had difficulty understanding the completion value, which made it hard for them to comprehend the purpose of the Do-expression.
Compared to experts the beginners didn't really noticed the small mistakes like not having quotation marks in the example in Listing 8. They did confuse it with the Do-while statement. In the beginning they thought it was a loop. They saw quickly that it couldn't be since their is no condition to end the loop.

Continuing from the previous response, the beginners had a harder time understanding the examples in the second part compared to the experts. They were not familiar with some of the syntax used in the examples and needed some explanation. They also had difficulty grasping the concept of the completion value and how it relates to the Do-expression.

In the third part, the beginners were more likely to confuse the Do-expression with the Do-while statement. They were not familiar with the Do-while statement either, but they associated the word "Do" with a loop construct. They had to be reminded that the Do-expression is not a loop construct and does not have a condition to end the loop.
Explaining them in the third part was a little harder, because they had never heard of completion value. First we had to go verbosely through completion and then describe what it would mean in the context of Do-expression. In this group everyone mentioned google as a search machine.

At the final part of the interview they did provide the correct output. The were not too curious about the limitations. When asked how they would solve it. All of them would not even use Do-expression to start with.
Since it was their first time they heard about the completion value, they didn't really see how this feature could be useful for them.

6 Conclusion

In conclusion, our survey findings provide valuable insights into the perspectives of both expert and beginner programmers regarding the Do-expression syntax in JavaScript. The interviews revealed that experienced programmers approach code differently, paying meticulous attention to details like missing semicolons, and possess a strong understanding of completion values. Importantly, all the experts unanimously regarded the Do-expression as a helpful feature that should be progressed in development.

On the other hand, beginners initially needed clarification, mistaking the Do-expression for a loop. However, they could comprehend its functionality after receiving clarification on its purpose. Despite this understanding, beginners were required to be more convinced of its practicality and expressed little intention to incorporate it into their work. The insights gathered from these interviews will significantly contribute to creating a comprehensive future survey better to understand programmers' perspectives on the Do-expression syntax.

Additionally, our experience underscores the importance of formulating clear and precise questions and providing well-crafted examples to minimize confusion and ensure that participants effectively grasp the research focus.

7 Discussion

This research aimed to assess the viability of the Do-expression syntax in JavaScript through interviews conducted with a limited sample size of six participants. However, it is crucial to gather a more extensive dataset by conducting a comprehensive survey to make a well-informed decision. Relying solely on the insights from six participants may not provide a sufficiently robust basis for determining the future of this feature.

Expanding the survey to include a more significant number of participants will yield more diverse perspectives and a broader range of experiences. By doing so, we can ensure a more representative sample, allowing for a more accurate assessment of the Do-expression syntax and its potential impact. Increasing the number of survey participants will enhance the findings' reliability and validity, thereby facilitating a more confident decision-making process.

A notable limitation of this research is that participants were presented with examples of the Do-expression syntax without the ability to execute the code. Programming languages are primarily designed to be executed, and evaluating code solely based on its static representation can pose challenges, particularly for beginners with limited experience in comprehending code structures effectively. To address this limitation, it is imperative to provide participants with the opportunity to execute the code and modify it according to their understanding and requirements.

By allowing participants to execute and manipulate the code, we can obtain more comprehensive and practical feedback on the Do-expression syntax. This approach will enable participants, especially beginners, to better understand the feature and provide more meaningful insights into its usability, potential challenges, and overall usefulness. Incorporating the execution and modification of code within the survey methodology will enhance the quality of feedback and contribute to a more thorough comprehension of the Do-expression syntax. By doing so, we can gather more valuable data that will aid in making informed decisions about this feature's future development and implementation.

References

[BG00] Alan F Blackwell and Thomas RG Green. A cognitive dimensions questionnaire optimised for users. In *PPIG*, volume 13. Citeseer, 2000.

[CH06] Kevin Crowston and James Howison. Hierarchy and centralization in free and open source software team communications. *Knowledge, Technology & Policy*, 18(4):65–85, 2006.

[Cla01] Steven Clarke. Evaluating a new programming language. In *PPIG*, volume 13, pages 275–289. Citeseer, 2001.

[CWHW08] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)*, 44(2):1–35, 2008.

[KLS16] Smitha Keertipati, Sherlock A Licorish, and Bastin Tony Roy Savarimuthu. Exploring decision-making processes in python. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*, pages 1–10, 2016.

[Lit18] Lia Litosseliti. *Research methods in linguistics*. Bloomsbury Publishing, 2018.

[SSS21] Pankajeshwara Nand Sharma, Bastin Tony Roy Savarimuthu, and Nigel Stanger. Extracting rationale for open source software development decisions—a study of python email archives. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 1008–1019. IEEE, 2021.

[TC3] Tc39: Specifying javascript.

[VSBS94] MW Van Someren, YF Barnard, and JAC Sandberg. The think aloud method: a practical approach to modelling cognitive. *London: AcademicPress*, 1994.