JOHANNES KEPLER
UNIVERSITY LINZ

Author
Lukas Tiefenthaler
k11907883

Submission
Institute for System Software

Thesis Supervisor
DI **Lukas Makor**

Assistant Thesis Supervisor
Dr. **Christian Wirth**

January 2023

Interoperability from java.time to Graal.js' Temporal

Bachelor's Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor's Program

Computer Science

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

Sworn Declaration

I hereby declare under oath that the submitted Bachelor's Thesis has been written solely by me without any third-party assistance, information other than provided sources or aids have not been used and those used have been fully documented. Sources for literal, paraphrased and cited quotes have been accurately credited.

The submitted document here present is identical to the electronically submitted text document.

Linz, January 2023

Lukas Tiefenthaler

Abstract

GraalVM allows embedding language interpreters like Graal.js into Java applications. Using GraalVM's Polyglot Interoperability features, the Java application can exchange data with the guest language. Using the Interop protocol, the guest language can query data of the host language (Java) similarly to querying native data of the guest language. An example is to read from a Java Array in code where a JavaScript Array is expected. That means a Java Array must be supported by the JavaScript code.

Both, Java Temporal (*java.time.**) and JavaScript Temporal, are APIs that contain types that describe date and time values. Furthermore, those libraries implement functionalities, which can be used to calculate using these values. For example, adding two durations together.

As part of this thesis, `java.time.*` should be implemented using a conversion, which can translate Java Temporal objects into Java time objects. To achieve this goal, these conversion methods must be implemented. They ensure, that the Java object, which should be supported by the JavaScript code is converted correctly, as well as helper functions, that determine which Temporal class object is needed.

Kurzfassung

GraalVM erlaubt es eingebettete Sprachinterpreter, wie Graal.js in Java Anwendungen zu benutzen. Mit der Benutzung von GraalVM's "Polyglot Interoperability"Feature kann die Java Anwendung Daten mit einer Gast Sprache austauschen. Man benutzt das İnterop"Protokoll, um Daten der Hauptsprache (Java) in einer Gastsprache anzufragen, ähnlich wie wenn man lokal Daten einer Gastsprache anfrägt. Ein Beispiel hierfür ist das lesen von einem Java Array, dort wo ein JavaScript Array erwartet wird. Das bedeutet, dass ein Java Array von einem JavaScript Code unterstützt werden muss.

Die Java Temporal (*java.time.**) und die JavaScript Temporal sind beides APIs, welche Typen enthalten mit welchen sie Datums- und Zeit-Werte beschreiben. Außerdem implementieren diese Bibliotheken Funktionalitäten, welche benutzt werden können, um mit diesen Werten zu rechnen. Zum Beispiel, das Addieren von von zwei Zeitspannen.

Als Teil dieser These soll *java.time.** mithilfe einer Konvertierungsfunktion implementiert werden, welche Java Temporal Objekte in Java time Objekte übersetzen kann. Um dieses Ziel zu erreichen, sind Umwandlungsmethoden implementiert worden. Diese versichern, dass das Java Objekt, welches von einem JavaScript Code unterstützt werden soll, richtig umgewandelt worden ist, sowie Hilfsfunktionen, welche feststellen, welches Temporal Klassenobjekt gebraucht wird.

Contents

Chapter 1

Introduction

GraalVM enables the integration of language interpreters like the Graal.js into Java applications. Using this technology in combination with GraalVM's Polyglot Interoperability features, an application, in this case a Java program can exchange data with a guest language, for example JavaScript. To do that, it uses the Interop protocol, which enables the guest language to query data from the host language (Java) similarly to querying native data of the guest language. For example, this process can be imaged as reading a Java array in code where a JavaScript array is expected. That means, that the Java array must be supported by the JavaScript code.

For this thesis, this process is applied to the Java Temporal *(java.time.*)* package, as well as its respective counterpart in JavaScript, the JavaScript Temporal package. Both of these packages describe date and time values, as well as functionalities, which are used to calculate with them. For example, calculating the duration between two points in time.

However, to make Java Temporal compatible with JavaScript Temporal, multiple conversion methods must be implemented, in order to make Java Code readable for the JavaScript language. These conversion methods must be able to link those two languages together and enable an exchange of data, as well as ensure, that they are converted correctly and without error. For that however, they need the correct input values, which have to be calculated differently for each Temporal class object. Furthermore, the conversion methods must be told which Temporal class object it is currently working on. For that, additional methods are required, which will be topics of this thesis.

It should be noted, that for this project, only the conversion methods, as well as case handling are part of this project. This thesis will only go into detail about those features, and only on a surface level explain other functionalities, or emit them completely, if deemed unnecessary. With this thesis, I hope to introduce the reader into the process of converting a Java object into a JavaScript object, so that a similar implementation can be replicated, using the same, or another Java library. Hence, at the end of this thesis, the reader should be able to implement something similar with ease.

The thesis is divided into four chapters, apart from the introduction. The first chapter are the *Fundamentals* and covers background information that is important for this thesis. In this chapter we are going into some detail about the components, as well as the basics needed to get an understanding about what this project entails. This information will mostly be on the surface level, and should only serve for a general understanding.

Chapter 3 is called *Interoperability support with java.time* and it goes into more detail about the project that was implemented. It serves to introduce the reader to the Temporal class objects, which have been implemented, as well as some of the algorithms that are used to make the project function. However, this chapter does not go into detail about how the project was implemented, but rather gives the reader a deeper understanding about what had to be implemented.

The implementation details are covered in Chapter 4. This is the main chapter of this theses, and it has the goal to show the reader how the project was implemented, using many examples. For the first few implementations, it goes into very much detail about

the process, however, many methods, which need to be extended in the implementation are essentially the same, with only minor changes. Therefore, as the chapter progresses, the descriptions will become less detailed, only describing new features.

The final one is Chapter 5, the *Testing* chapter. For this project, many tests have been written, in order to prove that the conversion is done as intended. This chapter covers all of them, however, in the same fashion as the previous chapter, will progressively be less detailed, as the tests are more or less the same.

Chapter 2

Fundamentals

Before we can get started with the task itself, let us first learn some of the most important fundamentals needed in order to understand how the GraalVM works. The topics that we are going to talk about are, GraalVM, ECMAScript, Truffle, Graal.js, as well as a short introduction to the Temporal Proposal.

2.1 GraalVM

GraalVM [3] is a project developed by Oracle. It was created based on the existing *Java Virtual Machine* (JVM) and promises to deliver better performance for Java applications. GraalVM entails a novel dynamic compiler that uses runtime information to perform extensive optimizations. With the use of a polyglot VM, apart from Java, other programming languages like JavaScript, Python, Ruby, or LLVM can be executed.

GraalVM runs Java code much more efficient than it is possible with the classic JVM. It achieves this by either interpreting the code directly, or by using a compiler. The latter is the most important functionality. It uses the *JVM Compiler Interface* to provide highly-efficient machine code for the compiled Java applications, in order to maximize its efficiency.

Additionally, GraalVM can also run Node.js, Ruby, R and Python. Therefore, these languages can also profit from the various optimizations performed by the GraalVM.

2.2 ECMAScript

ECMAScript is a standardized version of JavaScript, a programming language that is primarily used to create interactive front-end web applications. ECMAScript is developed and maintained by the ECMAScript International. It is yearly republished, with the latest version being ECMAScript 2022. It is a scripting language that conforms to the ECMAScript specification and is typically used to create client-side scripts for web browsers. JavaScript is an implementation of ECMAScript, and so it is fully compliant with the ECMA[1] specification. Furthermore, all its features are supported by GraalVM.

2.3 Truffle

The Truffle [8] framework allows to implement and run programming languages efficiently on GraalVM. It simplifies implementing language engines by automatically deriving high-performance code from interpreters. For that, it uses an abstract syntax tree, which enables almost any language to be adapted and runnable on the JVM. For Example, Ruby, R, and also JavaScript, which is the focus language in this thesis, can be adapted to run on the JVM.

2.3.1 Annotations

Truffle provides a handful of annotations, which are used for the implementation of this work. They are used a lot throughout every Truffle-based language interpreter to reduce boiler plate code and help the framework generate efficient code. In the next few subsections, we are going to describe the four most important annotations, which were needed in order to achieve the given task.

@ImportStatic

The *@ImportStatic* annotation imports all static functions of a file. It is defined as a class-level annotation and tells the class which methods it is allowed to work with. Additionally, it has a parameter, which numbers and configuration values are needed for it to work. For this thesis, the parameter is called *JSConfig.class*.

@TruffleBoundary

The *@TruffleBoundary* annotation marks a method which is considered a boundary for the Truffle partial evaluation which means it is not further inlined but treated as runtime call. This might be slightly slower regarding peak performance, but avoids code explosion especially around library methods.

@Specialization

This annotation defines a method of a node subclass to represent one specialization of an operation. Multiple specializations can be defined for each operation, and each defines which kind of input is expected, by using the method signature and the annotation attributes. This annotation has many constraints, however, in the scope of this task only one needs to be defined by the user, called *guard*.

The *guard* constraint is declared as a boolean expression, which defines whether a input is applicable to the specialization instance. For example, if the constraint is *@Specialization(guards = "interop.isInstant(thisObj)")*, only if this expression returns the value true, that underlying method will be called. Multiple guards can be defined at once, using a comma to separate them.

@CachedLibrary

The *@CachedLibrary* annotation is needed for allowing the use of Truffle libraries. It uses many constraints which define the behavior that the attached object should have, however, only one is important for the given task. The value that needs to be defined is the *limit* constraint, which defines the number of specialisations that are created before an uncached specialisation is used. This value is usually set to the string *InteropLibraryLimit*, a value that defines which class is used through *@ImportStatic*. Using this string, the compiler knows what function to call, the function that is defined with the correct limiter. In the scope of this task, this annotation is used in order to create the correct *JavaScript Temporal* class object.

2.4 Graal.js

Graal.js [2] is an JavaScript implementation, and is not only part of the GraalVM, but is actually one of its most complete guest language implementations to this point. Fitting

for the task of this project, Graal.js does support the execution of standalone JavaScript applications, as well as the execution of JavaScript from within a Java application. Additionally it comes with full support for Node.js and is compatible with the latest version of JavaScript's language specification, the ECMAScript Language Specification (Section 2.2).

Furthermore, not only does it support the interoperability with Java, but with other Truffle based languages as well, such as Ruby, Python, and R and uses a Truffle-based AST interpreter for its implementation. [9]. In total, the implementation effort is more than 80 thousand lines of code, with majority being written in Java, and only 1.5 thousand lines of code written in JavaScript.

2.5 ECMAScript Temporal proposal

The *ECMAScript Temporal proposal* provides standard objects and functions for working with dates and times. The proposal is currently in *Stage 3*, and has been a long-standing pain point in ECMAScript. It proposes `Temporal`, a global object, which acts as a top-level namespace (like `Math`), that brings a modern date/time API to the ECMAScript language. [5]

To give the reader a perspective how the following Temporal class objects are used, Figure 2.1 has been provided. It shows the relationship between the `Temporal` class objects and how they are constructed. A more detailed explanation, about what these Temporal class objects do is described in Chapter 3.

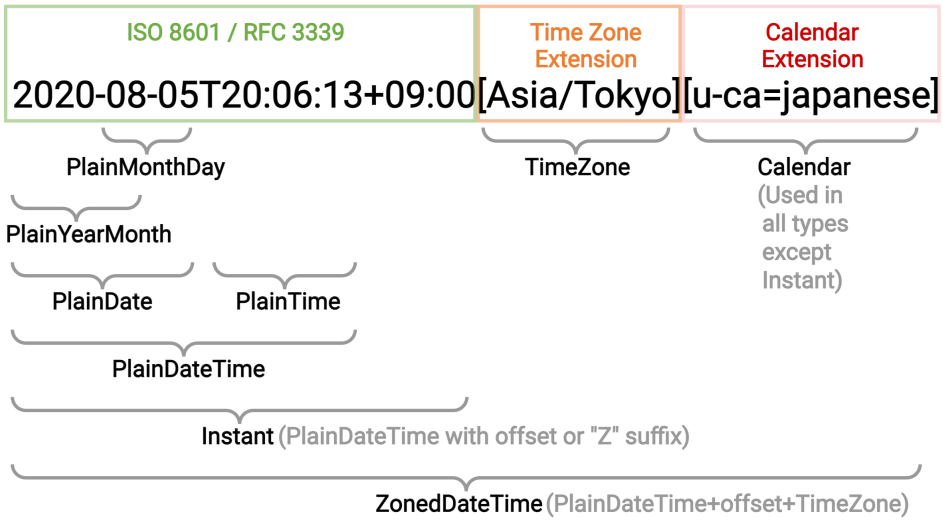

Figure 2.1: The correspondence between types and machine-readable strings [7]

Chapter 3

Interoperability support with java.time

This chapter is going into detail about the task that should be solved by this bachelor thesis. The task for this project was to implement the Interop feature for the JavaScript Temporal package. The goal of this is, that the *java.time* package which is used in Java for time conversion to be usable for the *JavaScript Temporal* package. To achieve this, some changes need to be done to the *GraalVM* repository.

As previously mentioned, the goal of this project is to convert a *java.time* class object into its *JavaScript Temporal* equivalent. A programmer who is operating in JavaScript code can use JavaScript Temporal objects, and using the GraalVM's polyglot feature, the user can even operate on Java Objects. In principle, a Java Temporal object should be usable when it arrives in JavaScript, however, in truth, JavaScript does not understand it, even if it is essentially the same. For example, A Java duration cannot be added to a JavaScript instant object under normal circumstances. That is exactly the task that needs to be solved in order to achieve this exact goal. A conversion must be written, that can transform a Java class object into JavaScript, so that it can be used everywhere. In Chapter 4, we are going into more detail how this is achieved.

3.1 Algorithm

The basic implementations for JavaScript class object methods have already been provided by GraalVM. However, changes need to be done in order for the code to recognize and work with the code that is provided in Java language. To achieve this, functions which calculate the correct values for the conversion must be implemented, which transform a Java object to its JavaScript counterpart.

Every Temporal class must at least have one such function, once for *Instant, Duration, PlainTime, PlainDate and TimeZone*. However, most of the time more than one such function needs to be implemented. That is because, sometimes case handling is not necessary, or is not wanted. For this project, usually three to four functions have been implemented, where three of them do essentially the same thing, checking if the function which is called meets all the requirements. Sometimes they do not need to meet all of them, which is why multiple such functions exist. The fourth function however is a bit special. This function is called *JavaInstantToInstant*, however, the names varies depending on the Temporal class which is used at the moment. These functions are responsible for converting the Java object classes to its JavaScript equivalents. In Chapter 4, we go more into detail how each of these functions are working and how they are implemented.

3.2 JavaScript Temporal class objects

In this section, we are going into more detail about the individual class objects of the *JavaScript Temporal* package. We are going to look at how to distinguish them and what special characteristics they have. Note that not every Temporal class is mentioned here, as in this project only a handful are implemented.

3.2.1 Instant

The *Temporal.Instant* defines a single point in time, called an exact time. It has a precision in nanoseconds. When a new Instant object is created, a parameter defining the time since the Unix time [10] must be defined in nanoseconds. However, an exact time can also be provided using the static function *Temporal.Instant.from()*. Using this expression, a point in time can be defined using an *ISO 8601* [4] formatted string that can be parsed and interpreted.

By Default, an Instant is interpreted in the UTC timezone. If it is interpreted in any other timezone, the value is shifted by a few hours. The nanoseconds representing the Instant object can be transformed into year, month, day, hours, minutes, seconds, milliseconds, microseconds and nanoseconds.

If an Instant is defined using the Unix time, a DateTime can be defined, that exists before *1970-01-01*. You can do that by using a negative value for the parameter of a new Instant. It should be noted, that Instant does not take leap-years into consideration.

3.2.2 Duration [6]

A *Temporal.Duration* object represents a time span. For example, one hour and 30 minutes. This value is represented using the *ISO-8601* standard notation. This notation starts with a *'P'* character, followed by year, month, week and day. Additionally, after that, it can have a *'T'* character, followed by hours, minutes, seconds, milliseconds, microseconds and nanoseconds. For example: *'P1Y1M1DT1H1M1.1S"* represents the duration: *one year, one month, one day, one hour, one minute, one seconds, and 100 milliseconds.* If a unit is omitted, it will be automatically set to zero. It can be noted, that a week can not be defined with other units, but must be defined all on its own.

This Duration has a special feature compared to every other *Temporal* class object. It does not balance its values automatically. That means it is possible to define two hours as 120 minutes, using the same notation: *'PT120M.* If you want to automatically balance these values, a method called *balancing* must be used.

Balancing

Unlike other Temporal types, the units in *Temporal.Duration* do not wrap around zero, because you may want to get a duration in a specific time unit.

Every other Temporal type has a maximum for each time unit. By default, all of them use the balancing constraint mode by default. That means that every value above the maximum number will be replaced with the maximum value instead.

If Duration must be balanced, a couple of methods can be applied. For example:

- **Rounding**
  The first method is the rounding method. Here the round function is used on the duration. Round does have a option called *largestUnit*. Setting this value to a specific unit, for example hours, balances the whole expression to hours, but not taking units greater then hours into account (Listing 3.1).

Listing 3.1: Rounding Duration

```
1 d = Temporal.Duration.from({ minutes: 80, seconds: 90 }); // => PT80M90S
2 d.round({ largestUnit: 'hour' }); // => PT1H21M30S (fully balanced)
```

- **Serialization**
  Another method, which works for every Temporal class object is the serialize to a

string, using the *toString()* method, and deserializing it again by calling *from()* on the string.

This works correctly for every unit in duration, except milliseconds, microseconds and nanoseconds. If those values are greater than 999, the duration will yield an incorrect value. This has to do with the way those units are implemented and will be described in Chapter 4

3.2.3 TimeZone

A *Temporal.TimeZone* object is the representation of a time zone. It uses the *IANA time zone database* to get information about the time zone, like the offset between the local time and UTC at a particular time, *daylight saving time* (DST), as well as other political UTC offsets.

For example, this class can be used in order to be able to convert a *Temporal.Instant* object into a *Temporal.PlainTime* object.

3.2.4 PlainTime

A *Temporal.PlainTime* represents a wall-clock time, and has a precision in nanoseconds. It is not associated with any time zone. Wall-clock time refers to the concept of a time which is expressed in everyday usage. For example, a PlainTime object can look like: *15:30:00*, which is easily distinguishable as half past three. It should be noted, that PlainTime is not associated with any specific day, or timezone. It only describes a wall-clock time.

3.2.5 PlainDate

A *Temporal.PlainDate* object defines a calendar date, where *Calendar date* refers to the concept of a date expressed in everyday usage. It is independent of any time zone and always represents the whole day. For example, a day can be represented as *'2020-03-14'*.

Chapter 4

Implementation

In this chapter we are going to describe how the problem in Chapter 3 was solved. To achieve this, a couple of classes must be changed in order for it to function as intended. Usually, every JavaScript Temporal class object has its own class in which it is handling every builtin methods for its class object. For a concrete transformation, a conversion method had to be implemented for each Java Temporal type. The respective methods read information from the Java type and create a new object from the set of available JavaScript Temporal types, that represent the same value as the Java type. Those conversion methods are then used in the specializations of the Temporal builtin methods that need to accept those Java types. They make sure that, in case a Java Temporal type is provided, it is converted to a JavaScript Temporal type. The rest of the builtin method can thus remain unmodified, as only JavaScript temporal types are allowed to flow any further into those methods.

4.1 Setting up the environment

In order to get started with this project some actions have to be taken to setup the project properly on your local system. Oracle has provided a step to step guide on how to get started using the GraalVM repository. It is recommended to be using a Linux operating system or a virtual machine that is capable of running a Linux systems, such as Ubuntu. Similar operating system can be used, such as Windows and Mac, however, that is not recommended, because setting up the project in these is much more difficult compared to Linux. A guide for the Linux installation can be found on the official GraalVM git repository page.

4.2 Conversion helper methods

4.2.1 java.time.Instant to Temporal.Instant

The class *TemporalInstantPrototypeBuiltins* provides the methods of the *Temporal.Instant* type as previously introduced in Chapter 3. For this class, a couple of methods need to be implemented. Most of these methods, that are representing the JavaScript Temporal methods will be introduced in Section 4.3.1. However, another function needs to be implemented before the conversion can work properly.

Listing 4.1 shows how the function that converts a *java.time.instant* into a *JavaScript Temporal.Instant* is implemented. It can be observed, that the Java instant has been multiplied by one billion. That's because the Java object is measured in the unit second, and its JavaScript counterpart uses the unit nanoseconds. Therefore, a multiplication of one billion is necessary, in order to get the correct value. Additionally, the nanoseconds from the Java Instant must be added separately, because *instant.getEpochSeconds()* only has a precision accurate enough to get microseconds.

After the conversion is completed, a new *JSTemporalInstant* object can be created. Should the calculation fail, because the input is not a Instant object, *null* should be returned, indicating that the object could not be created.

Listing 4.1: The method that converts a Java java.time.Instant object to a JavaScript Temporal Instant object

```
1  @TruffleBoundary
2  public static JSTemporalInstantObject javaInstantToInstant(Object thisObj,
       InteropLibrary interop, JSContext context) {
3      try {
4          java.time.Instant instant = interop.asInstant(thisObj);
5          BigInteger bi = BigInteger.valueOf(instant.getEpochSeconds());
6          bi = bi.multiply(BigInteger.valueOf(1_000_000_000));
7          bi = bi.add(BigInteger.valueOf(instant.getNano()));
8          return JSTemporalInstant.create(context, new BigInt(bi));
9      }
10     catch (UnsupportedMessageException e) {
11         return null;
12     }
13 }
```

4.2.2 java.time.Instant to Temporal.Duration

The class *TemporalDurationPrototypeBuiltins* provides the methods of the *Temporal.Duration* type as previously introduced in Chapter 3. Similarly to the previous class, this class must be expanded identically. Likewise, this will also be described in Section 4.3.2 in more detail. However, yet another function must be implemented, in order to convert the Java Duration into its JavaScript equivalent.

Listing 4.2 shows how the function that converts a *java.time.duration* into a *JavaScript Temporal.Duration* is implemented. Similarly to Listing 4.1, the Java Duration class does have the correct units available. Functions like the *toSeconds()* or *toMinutes()* do exist. However, they should rather not be used, as the result can be not what is desired. For example, two days are correctly saved as two days, however, they are also saved as 48 hours, 2880 minutes, and so on. It makes it difficult to determine which unit is originally meant. In order to achieve this, other methods must be used, which luckily have already been provided by GraalVM. The *getSeconds()* and *getNano()* methods does, in a way, what we want it to do.

For example, the *getNano()* function returns a value which is a combination of milliseconds, microseconds and nanoseconds into a value, where the three right most digits represent the nanoseconds, the middle three represent the microseconds, and the three left most the milliseconds. For example, the number *123456789* would represent 123 milliseconds, 456 microseconds and 789 nanoseconds. That can be observed in Listing 4.2, line 12-14.

getSeconds() works in a similar fashion, but needing a different calculation to get individual unit values. Using the common calculation used for converting between hours, minutes and seconds, as observed in Listing 4.2, line 9-11, we can see how each unit has its value calculated, without having the trouble of getting the wrong values.

It should be noted, that the values years, months, and weeks are not calculated, but instead specified as zero. That is because, the Java Duration class does not handle these values. The Period class is responsible for them, but was omitted, as it was not deemed necessary for the conversion of Java Duration.

Now that the conversion is completed, a new *JSTemporalDuration* object can be created. Should the calculation fail, because the input is not a Duration object, *null* should be returned.

Listing 4.2: The method that converts a Java java.time.Duration object to a JavaScript Temporal Duration object

```
1  @TruffleBoundary
2  public static JSTemporalDurationObject javaDurationToDuration(Object thisObj
       , InteropLibrary interop, JSContext context) {
3      try {
4          java.time.Duration duration = interop.asDuration(thisObj);
5          double days = duration.toDays();
6          double years = 0;
7          double months = 0;
8          double weeks = 0;
9          double hours = Maths.floor(duration.getSeconds() / 3_600) % 24;
10         double minutes = Maths.floor(duration.getSeconds() / 60) % 60;
11         double seconds = duration.getSeconds() % 60;
12         double milsec = Math.floor(duration.getNano() / 1_000_000) % 1_000;
13         double microsec = Math.floor(duration.getNano() / 1_000) % 1_000;
14         double nanoseconds = duration.getNano() % 1_000;
15
16         return JSTemporalDuration.createTemporalDuration(context, years,
       months, weeks, days, hours, minutes, seconds, milsec, microsec,
       nanoseconds);
17     }
18     catch (UnsupportedMessageException e) {
19         return null;
20     }
21 }
```

4.2.3 java.time.Instant to Temporal.TimeZone

The class *TemporalTimeZonePrototypeBuiltins* provides the methods of the *Temporal.TimeZone* type as previously introduced in Chapter 3. Likewise, this class must be expanded in a similar fashion as the previous classes. The available functions, which need to be implemented in this class are described in Section 4.3.3 in more detail. Similarly to the former classes, this class also has a function called *javaTimeZoneToTimeZone* which needs to be implemented, in order to convert a *java.time.ZoneId* object to its *JavaScript Temporal.TimeZone* equivalent.

Listing 4.3 the function that is responsible for converting a *Java ZoneId* object into a *JavaScript TimeZone* object. For this conversion, only one step is necessary. A *TruffleString* must be created using the function in Listing 4.3, line 5.

Now that the conversion is completed, a new *JSTemporalTimeZone* object can be created. Should the calculation fail, because a TimeZone object could not be created, *null* should be returned.

Listing 4.3: The method that converts a Java java.time.ZoneId object to a JavaScript Temporal TimeZone object

```
@TruffleBoundary
public static JSTemporalTimeZoneObject javaTimeZoneToTimeZone(Object thisObj
    , InteropLibrary interop, JSContext ctx) {
    try {
        java.time.ZoneId zone = interop.asTimeZone(thisObj);
        TruffleString identifier = TruffleString.fromJavaStringUncached(zone
    .getId(), TruffleString.Encoding.UTF_32);

        return JSTemporalTimeZone.create(ctx, null, identifier);
    }
    catch (UnsupportedMessageException e) {
        return null;
    }
}
```

4.2.4 java.time.Instant to Temporal.PlainTime

The class *TemporalPlainTimePrototypeBuiltins* provides the methods of the *Temporal.PlainTime* type as previously introduced in Chapter 3. The available functions, which need to be implemented in this class are described in Section 4.3.4 in more detail. Similarly to the previous classes, this class also needs to implement a function, which allows a correct conversion of the *Java LocalTime* to its counterpart, the *JavaScript PlainTime*. The function is called *javaPlainTimeToPlainTime*, and it can be implemented almost identical to the class handling Duration objects, however requiring an additional parameter of the *BranchProfile* class.

Listing 4.4 shows how the function is implemented, which is converting a *Java LocalTime*, into a *JavaScript PlainTime* object. Similarly to the Duration class, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds must be extracted from the LocalTime object. However, PlainTime works differently from Duration. Duration, in contrast, does by default not use the balancing feature, meaning that the values of each unit do not round to zero. More on that topic can be read in Section 3.2.2. Every other *JavaScript Temporal* class object does implement balancing by default. That means, that it is unnecessary to take those value overflows into consideration, as for example *time.getMinute()* can only be in the range of 0-59. That means, that if you want to illustrate 120 minutes, it would automatically be converted into two hours and zero minutes.

It can be observed in Listing 4.4, line 8-10, that milliseconds, microseconds, and nanoseconds are still converted in the same way the Duration class implementation would do. This is a design choice, that was provided as such by *GraalVM* and does not have anything to do with balancing.

Now that the conversion is completed, a new *JSTemporalPlainTime* object can be created. Should the calculation fail, because a PlainTime object could not be created, *null* should be returned.

Listing 4.4: The method that converts a Java java.time.LocalTime object to a JavaScript Temporal PlainTime object

```
1  @TruffleBoundary
2  public static JSTemporalPlainTimeObject javaPlainTimeToPlainTime(Object
       thisObj, InteropLibrary interop, JSContext context, BranchProfile
       errorBranch) {
3      try {
4          java.time.LocalTime time = interop.asTime(thisObj);
5          int hours = time.getHour();
6          int minutes = time.getMinute();
7          int seconds = time.getSecond();
8          int milsec = (int) Math.floor(time.getNano() / 1_000_000) % 1_000;
9          int microseconds = (int) Math.floor(time.getNano() / 1_000) % 1_000;
10         int nanoseconds = time.getNano() % 1_000;
11
12         return JSTemporalPlainTime.create(context, hours, minutes, seconds,
       milsec, microseconds, nanoseconds, nanoseconds, errorbranch);
13     }
14     catch (UnsupportedMessageException e) {
15         return null;
16     }
17 }
```

4.2.5 java.time.Instant to Temporal.PlainDate

The class *TemporalPlainDatePrototypeBuiltins* provides the methods of the *Temporal.PlainDate* type as previously introduced in Chapter 3. The available functions, which need to be implemented in this class are described in Section 4.3.5 in more detail. Similarly to the previous classes, this class also needs to implement a function, which allows a correct conversion of the *Java LocalDate* to its counterpart, the *JavaScript PlainDate*. The function is called *javaPlainDateToPlainDate*, and it is essentially the same as the PlainTime class, but it does not convert the wall-clock units, like hours, minutes, seconds, etc., but instead converting the units day, month, and year.

Listing 4.5 shows how the function is implemented, which is converting a *Java LocalDate*, into a *JavaScript PlainDate* object. As mentioned in the previous paragraph, *PlainDate* is similar to *PlainTime* implementation, with the only major difference being, that instead of wall-clock units, year, month and day are converted. As it can be observed in Listing 4.5, line 5-7, the *Java LocalDate* object does already have the correct values for this conversion. That makes it quite easy to convert, as it only needs to be copied.

However, an additional value is needed, a JSDynamicObject, which must fetch the correct calendar type. Listing 4.5, line 10, shows exactly how this is done, using a context and realm, which can simply be retrieved by using the *getRealm()* method.

Now that the conversion is completed, a new *JSTemporalPlainDate* object can be created. Should the calculation fail, because a PlainDate object could not be created, *null* should be returned.

Listing 4.5: The method that converts a Java java.time.LocalDate object to a JavaScript Temporal PlainDate object

```
1  @TruffleBoundary
2  public static JSTemporalPlainDateObject javaPlainDateToPlainDate(Object
       thisObj, InteropLibrary interop, JSContext context, JSRealm realm) {
3      try {
4          java.time.LocalDate date = interop.asDate(thisObj);
5          int year = date.getYear();
6          int month = date.getMonthValue();
7          int day = date.getDayOfMonth();
8          JSDynamicObject calendar = TemporalUtil.getISO8601Calendar(context,
       realm);
9
10         return JSTemporalPlainDate.create(context, year, month, day,
       calendar);
11     }
12     catch (UnsupportedMessageException e) {
13         return null;
14     }
15 }
```

This class has an additional feature, which the other classes do not have. It collectively is used by all the previously mentioned classes, to implement functions that help with the conversion to the *JavaScript Temporal* class objects, primarily with the beforehand mentioned exception handling. For the next few examples, code snippets for the *Temporal Instant* class object will be shown, however, each of the other classes are an almost identical copy of these.

Listing 4.6 shows the first method, which is needed for each and every Temporal class. These functions simply tests if the input obj is already the correct *JavaScript Temporal* class object. This function is a basic version of Listing 4.7, and is rarely used.

Listing 4.6: The function that tests if the input is already the correct JavaScript Temporal class object

```
1  protected JSTemporalInstantObject requiresTemporalInstant(Object obj) {
2      if (!(obj instanceof JSTemporalInstantObject)) {
3          errorBranch.enter();
4          throw TemporalErrors.createTypeErrorTemporalInstantExpected();
5      }
6      return (JSTemporalInstantObject) obj;
7  }
```

Listing 4.7 is an advanced version of the previous shown function. It also tests the input object, if it is already of the correct *JavaScript Temporal* class. If that is not the case, it does not immediately throw an exception, but uses the additional parameter *InteropLibrary* to test, if the object can be converted into the correct *JavaScript Temporal* class object. If that is the case, this function calls the function, that converts them to the correct type, as shown in Listing 4.7, line 6. Similarly to the basic function, should both cases be false, an exception should be thrown.

These methods need to be used for every *JavaScript Temporal* conversion method at least once, as it not only tests the validity of the input obj, but also calls the function that converts it correctly on it.

Listing 4.7: The function that tests if the input is already the correct JavaScript Temporal class object and converts the input if it is not the case

```
protected JSTemporalInstantObject requiresTemporalInstant(Object obj,
    InteropLibrary interop, JSContext ctx) {
    if (obj instanceof JSTemporalInstantObject) {
        return (JSTemporalInstantObject) obj;
    }
    if (interop.isTime(obj)) {
        JSTemporalInstantObject inst = TemporalInstantPrototypeBuiltins.
    JSTemporalInstantGetterNode.javaInstantToInstant(obj, interop, ctx);
        return inst;
    }
    errorBranch.enter();
    throw TemporalErrors.createTypeErrorTemporalInstantExpected();
}
```

The final method, as shown in Listing 4.8, does exactly the same as the previous method, however, it does not throw an exception, if the object could not be converted correctly. That is because this function is only used, when another method already handles the exceptions. Therefore, it is not needed to check multiple times.

Listing 4.8: The function that tests if the input is already the correct JavaScript Temporal class object but does not throw an exception

```
protected JSTemporalInstantObject convertJavaToJavascriptInstant(Object obj,
     InteropLibrary interop, JSContext ctx) {
    if (interop.isTime(obj)) {
        JSTemporalInstantObject inst = TemporalInstantPrototypeBuiltins.
    JSTemporalInstantGetterNode.javaInstantToInstant(obj, interop, ctx);
        return inst;
    }
    return (JSTemporalInstantObject) obj;
}
```

The function, which ensures that every conversion only calls object with the correct type. In the scope of this project, that would be *Instant, Duration, TimeZone, PlainTime, and PlainDate*. In the next sections of this chapter, we are going into more detail how the *JavaScript Temporal* class methods are implemented.

4.3 Builtin methods

Builtin methods provide the methods of the object of the respective class. For example, *Temporal.Instant.prototype.add* provides the methods myInstant.add..., where myInstant is a Temporal.Instant class object.

4.3.1 Temporal Instant

In this section we are going into more detail about the methods implemented in *Temporal.Instant* class. Those methods include:

- **Temporal.Instant.prototype.add**
  One of the simplest methods, which need to be implemented is the *add* function. This method is described first as it can be a simple template for all further methods discussed. Listing 4.9 shows how the class in which the method is defined is implemented, and analog to it, how every other methods in this file, as well as in other files should be implemented.

The class is an abstract static class, which extends the *JSTemporalBuiltinOperation* class. It uses an *@ImportStatic* annotation, which has already been explained in Section 2.3.1. The class itself has two functions defined. A constructor, which simply makes a super call, as well as the actual add method, which uses yet another annotation called *@Specialization*. For more detail on this annotation, see Section 2.3.1

Going into more detail how the add is implemented, we can observe, that the method has some interesting parameters. The object *thisObj*, which represents the receiver *("this")* of the function, that has been called. In this case, the add has been called. This object usually is of the type *Temporal.Instant*. However, in the scope of this project, *java.time.Instant* is used, and almost never the most common type *Temporal.Instant*, in order to test if the conversion of these two types is possible. In the latter case, as shown in Listing 4.7, line 2, the object will simply be returned, without doing anything with it, as it is the expected type.

The second parameter, in this case, is another object, called *temporalDurationLike*. This object must be provided as a argument in the function call, and is later used to increment *thisObj* by the value provided in this parameter. This object must be of type *Temporal.Duration*, as the conversion of a *Java.Duration* is not supported in this function.

The third parameter, which does not need to be passed, but is automatically generated instead, is a new *ToLimitedTemporalDurationNode* object, which is used in order to create a *JSTemporalDurationRecord* object. Using this object, the exact duration, that needs to be added to *thisObj* can be extracted, as shown in Listing 4.9, line 13-14.

The last parameter for this add function is a *InteropLibrary* object. This object is used, in order to determine which *JavaScript Temporal* object should be expected. In Section 4.4, we go into more detail, how this process is implemented.

Finally, if the conversion of the Instant object and extraction of the Duration object are completed, *TemporalUtil.addInstant()* is called, which increments the Instant by the value of the Duration, and creates a new *JSTemporalInstant* object, as shown in line 14-15.

It should be noted, that most of the mentioned functionality has been provided prior. The task was the conversion of the types, and therefore the use of the interop parameter, and in consequence every function that makes use of this argument. For example the Listing 4.7, as well as Listing 4.8. The same principle also holds for every other function mentioned in this chapter, and will therefore not be mentioned additionally.

Listing 4.9: The method that adds two Temporal.Instant values together

```
1  @ImportStatic({JSConfig.class})
2  public abstract static class JSTemporalInstantAdd extends
       JSTemporalBuiltinOperation {
3
4      protected JSTemporalInstantAdd(JSContext context, JSBuiltin builtin)
       {
5          super(context, builtin);
6      }
7
8      @Specialization
9      public JSDynamicObject add(Object thisObj, Object
       temporalDurationLike,
10                                @Cached("create()")
       ToLimitedTemporalDurationNode toLimitedTemporalDurationNode,
11                                @CachedLibrary(limit = "
       InteropLibraryLimit") InteropLibrary interop) {
12         JSTemporalInstantObject instant = requireTemporalInstant(thisObj,
            interop, getContext());
13         JSTemporalDurationRecord duration = toLimitedTemporalDurationNode
       .executeDynamicObject(temporalDurationLike, TemporalUtil.
       listPluralYMWD);
14         BigInt ns = TemporalUtil.addInstant(instant.getNanoseconds(),
       duration.getHours(), duration.getMinutes(), duration.getSeconds(),
       duration.getMilliseconds(), duration.getMicroseconds(), duration.
       getNanoseconds());
15         return JSTemporalInstant.create(getContext(), getRealm(), ns);
16     }
17 }
```

- **Temporal.Instant.prototype.subtract**
  The subtract class is called *JSTemporalInstantSubtract*, and it is not very different from the previous add. The only difference between these two classes is, that if we look at Listing 4.9, line 14, every duration call has its sign changed. For example, *duration.getHours()* becomes *-duration.getHours()*, and hence forth.

- **Temporal.Instant.prototype.until and .since**
  The until and since functions are, however a bit different. First and foremost, both of these *JavaScript Temporal.Instant* functions are combined into one, meaning both use the same function to determine their results. To differentiate between those two cases, the function utilizes a boolean called *isUntil*, which indicates, if the should be handled as an until, or as a since.

  Additionally, this function, which is called *untilOrSince*, uses different parameters. The parameters *thisObj*, and *InteropLibrary* are still used, as they must be used for every function, however, another object called *otherObj* is used. This parameter functions exactly like *thisObj*. However, as shown in Listing 4.10, a different method is called (Listing 4.8). The function also has an optional parameter called *optionsParam*, which can define modes, like defining the largest/smallest unit which is allowed to be returned, and so on.

  The other parameters are called *JSToNumberNode, EnumerableOwnPropertyNamesNode, ToTemporalInstantNode, and TruffleString.EqualNode*, which are used to call other functions on, and will not be discussed further, as they are not part of the implementation.

Listing 4.10: UntilOrSince code snippet

```
1 JSTemporalInstantObject instant = requireTemporalInstant(thisObj, interop
      , getContext());
2 JSTemporalInstantObject other = toTemporalInstantNode.execute(
      convertJavaToJavascriptInstant(otherObj, interop, getContext()));
```

- **Temporal.Instant.prototype.round**
  The round class is called *JSTemporalInstantRound*. It has a function called *Round* and only needs an additional parameter, called *roundToParam*, which is used to define, to which unit - hour, minute, second, millisecond, microsecond or nanosecond - the Instant object should be rounded to. By default, the round function uses a half expand. The function returns a Instant object, which will be rounded to the unit defined in the parameter *roundToParam*.

- **Temporal.Instant.prototype.equals**
  The equals class is called *JSTemporalInstantEquals*. It defines the function *Equals*, and once again uses the function described in Listing 4.8 to convert *Java Instant* to *JavaScript Instant*. This function returns a boolean, which is either true, or false, depending if the *thisObj* and *otherObj* are the same.

- **Temporal.Instant.prototype.toString**
  The toString function, in the class *JSTemporalInstantToString*, once again uses an additional parameter called *optionsParam*, where for example, the timezone can be defined. The parameter *thisObj* should be a Instant object, which is defined as a number that defines the time that has past since 1970-01-01. The function then converts the number into a semi-readable format and returns this as a string.

- **Temporal.Instant.prototype.toLocaleString**
  The ToLocaleString function, which is implemented in the class *JSTemporalInstant-ToLocaleString*. It converts the input object into a format, that is readable by a human. For example, *2019-11-18T11:00:00.000Z* could be converted into the format *2019-11-18, 3:00:00 a.m.*, depending on the location that the user is currently at.

- **Temporal.Instant.prototype.toJSON**
  This function is not implemented

- **Temporal.Instant.prototype.valueOf**
  The function valueOf is implemented in the class *JSTemporalInstantValueOf*. This function only throws a exception if it is called, because, by default, valueOf should not be supported by the *Temporal.Instant* class.

- **Temporal.Instant.prototype.toZonedDateTime**
  The function ToZonedDateTime is implemented in the class *JSTemporalInstantTo-ZonedDateTimeNode*. The function has, next to the regular parameters a parameter called *item*. This item object defines a calendar, as well as a timezone. The calendar is defined as a *Temporal.Calendar* object and set to a handful of calendar-types like *gregory*, or *japanese*. The timezone, that needs to be defined, can be chosen from a list described in the *IANA time zone database*. After everything has been calculated, the function will return a *JSTemporalZonedDateTime* object.

- **Temporal.Instant.prototype.toZonedDateTimeISO**
  The function ToZonedDateTimeISO is implemented in the class *JSTemporalInstant-ToZonedDateTimeISONode*. Additionally to the normal parameters, it also has an parameter called *itemParam*. This object is a timezone object which defines a time-zone described in the *IANA time zone database*. The only difference between this function and the function described in the previous bullet point is, that by default the calendar ist set to *ISO-8601*. This function will also return a *JSTemporalZoned-DateTime* object.

4.3.2 Temporal Duration

In this section we are going into more detail about the methods implemented in *Temporal.Duration* class. Those methods include:

- **Temporal.Duration.prototype.with**
  The with function is the most basic function of the *Temporal.Duration* class, and works similarly as the *Temporal.Instant* class. It once again uses a receiver called *thisObj*, which in the scope of this project is usually of the type *java.time.Duration*. Once again it will be converted using the function *requireTemporalDuration(thisObj, interop, getContext())*. The return value for this function is a perfect copy of this converted value.
- **Temporal.Duration.prototype.add**
  The add function works quite similarly to the function mentioned in *Temporal.Instant*, however it uses a second Duration object as a parameter, which needs to be converted as well, using the usual *requireTemporalDuration(thisObj, interop, getContext())* for *thisObj*, but additionally call the function *convertJavaToJavascriptDuration(otherObj, interop, getContext())*. Now the add function must get the sum of both durations and return a this new Duration object.
- **Temporal.Duration.prototype.subtract**
  The subtract function is essentially the same as the add function, described in the previous paragraph, however, after converting both parameters, the second argument must be subtracted. In the case of this project, a sign change has been used.
- **Temporal.Duration.prototype.negated**
  The negated function is quite the same as the previously presented function with. However, instead of returning a perfect copy right away, it first negates every unit in the duration. Once again using the *requireTemporalDuration(thisObj, interop, getContext())* to convert the function successfully.
- **Temporal.Duration.prototype.abs**
  The abs function will always return a positive function. It first converts the *java.time.Duration* type into a *Temporal.Duration* object and uses the *Math.abs* function on it, before returning it as a Duration object.
- **Temporal.Duration.prototype.round**
  The round function is implemented in the class *JSTemporalDurationRound*. The function has a receiver called *thisObj*, which must be a Duration object. Additionally, a option parameter must be present, that determines to what unit the duration value should be rounded to. The round method uses the half expand rounding mode. Similarly to the previous function, *thisObj*, must be converted, if the input class is not of type *Temporal.Duration*. The return value will be a *JSTemporalDuration* object.
- **Temporal.Duration.prototype.total**
  The total function is implemented in the class *JSTemporalDurationTotal*, and has a Duration object as a receiver, as well as a parameter called *totalOfParam*, which determines to which unit the value should be converted. For Example, if that value is set as seconds, and the input is two minutes, the result should be a *JSTemporalDuration* object. Similarly, a conversion of the receiver object must be done.
- **Temporal.Duration.prototype.toString**
  The toString function is implemented in the class *JSTemporalDurationToString*, and converts units of durations into the compact form for duration. For example, 10 hours would be converted into *PT10H*. Once again, *thisObj* must be converted into a *Temporal Duration* object. The function returns a string object, as described.
- **Temporal.Duration.prototype.toLocaleString**
  The toLocaleString function, whích is implemented in the *JSTemporalDurationToLocaleString* class works identically to the toString function. It converts the parameter *thisObj* and returns a compact string object as described in the previous bullet point.

- **Temporal.Duration.prototype.toJSON**
  This function is not implemented
- **Temporal.Duration.prototype.valueOf**
  The function valueOf is implemented in the class *JSTemporalDurationValueOf*. This function only throws a exception if it is called, because, by default, valueOf should not be supported by the *Temporal.Duration* class.

4.3.3 Temporal TimeZone

In this section we are going into more detail about the methods implemented in *Temporal.TimeZone* class. Those methods include:

- **Temporal.TimeZone.prototype.getOffsetNanosecondsFor**
  The getOffsetNanosecondsFor function is implemented in the class *JSTemporalTimeZoneGetOffsetNanosecondsFor*. It has a receiver object called *thisobj*, which must be a timezone object. This object is converted using the *requireTemporalTimeZone* function. Additionally, the function has a parameter called *instantParam*, which is a *Temporal.Instant* class object, which is used to compute the time zone's UTC offset. The function returns this calculated UTC offset in nanoseconds.
- **Temporal.TimeZone.prototype.getOffsetStringFor**
  The function getOffsetStringFor is implemented in the class *JSTemporalTimeZoneGetOffsetStringFor*, and has, similarly to the previous mentioned function a receiver timezone object, which needs to be converted as well as a *Temporal.Instant* class object that computes the time zone's UTC offset. However, the function does not return a value in nanoseconds, but converts it into a string object, like *"-08:00"*, which indicates a eight hour negative offset.
- **Temporal.TimeZone.prototype.getPlainDateTimeFor**
  The getPlainDateTimeFor function is implemented in the class *JSTemporalTimeZoneGetPlainDateTimeFor*. It has a receiver timezone object, that needs to be converted, as well as a *Temporal.Instant* object, that indicates an exact point in time. Additionally, the function has an optional calendar parameter, that by default uses *ISO-8601*. The function returns a *Temporal.PlainDateTime* class object, which represents the date, time and timezone of the provided Instant object.
- **Temporal.TimeZone.prototype.getInstantFor**
  The getInstantFor function is implemented in the class *JSTemporalTimeZoneGetInstantFor*. This function is the counterpart to the function mentioned in the previous function. It has a timezone receiver object that it converts into a *Temporal.TimeZone* object, but instead of transforming a Instant object into a PlainDateTime object, it does the opposite and transforms a parameter representing a datetime object into a instant object.
- **Temporal.TimeZone.prototype.getPossibleInstantsFor**
  The function getPossibleInstantsFor is implemented in the class *JSTemporalTimeZoneGetPossibleInstantsFor*. It is almost indentical to the previous function, however, it does not have a optional parameter, that could be used to configure the calculation.
- **Temporal.TimeZone.prototype.getNextTransition**
  The getNextTransition function is implemented in the class *JSTemporalTimeZoneGetNextTransition*. It has a receiver timezone object, that needs to be converted into a *Temporal.TimeZone* object, as well as a *Temporal.Instant* object argument. The function returns the next datetime object, which represents the next time that the offset of the specified timezone changes. For example, when the when a time change occurs.

- **Temporal.TimeZone.prototype.getPreviousTransition**
  The getPreviousTransition function is implemented in the class *JSTemporalTimeZoneGetPreviousTransition*, and equally to the previous function returns a datetime object. This object, however, represents the last time the offset for the specified timezone has changed.
- **Temporal.TimeZone.prototype.toString**
  The toString function is implemented in the class *JSTemporalTimeZoneToString*, and converts a *timezone.id* object into a human-readable description of the timezone.
- **Temporal.TimeZone.prototype.toJSON**
  The toJSON function is actually implemented in the class *JSTemporalTimeZoneToJSON*, however, it is the same as the toString function, and should like for the other classes not be called directly.
- **Temporal.TimeZone.prototype.valueOf**
  The function valueOf is implemented in the class *JSTemporalTimeZoneValueOf*. This function only throws a exception if it is called, because, by default, valueOf should not be supported by the *Temporal.TimeZone* class. Note that this function does not exist in the official *Temporal.TimeZone* implementation, but is still supported by Graal.js.

4.3.4 Temporal PlainTime

In this section we are going into more detail about the methods implemented in *Temporal.PlainTime* class. Those methods include:

- **Temporal.PlainTime.prototype.with**
  The function with is implemented in the class *JSTemporalPlainTimeWith*. It has a PlainTime object as a receiver, which needs to be converted into a *Temporal.PlainTime* object. Additionally, it has a parameter called *timelike*, which holds some, or all properties of the PlainTime object. For example, the PlainTime object could hold the time *"15:30:00"*. The timelike argument could now set the minute unit to zero. In that case, the function would return a plaintime object that has, if converted to a string, the value *"15:00:00"*.
- **Temporal.PlainTime.prototype.add**
  The function add is implemented in the class *JSTemporalPlainTimeAdd*, and adds a duration to a the receiver PlainTime object, which must be converted inside the function. A new *JSTemporalPlainTime* object is returned at the end of the function.
- **Temporal.PlainTime.prototype.subtract**
  The function subtract is implemented in the class *JSTemporalPlainTimeSubtract*. It works in a similar fashion as the previous mentioned function. But instead of simply adding a duration to a PlainTime object, it first changes the sign of the duration, because adding it. The result is yet again a new *JSTemporalPlainTime* object.
- **Temporal.PlainTime.prototype.until**
  The until function is implemented in the class *JSTemporalPlainTimeUntil*. It has a PlainTime receiver object, as well as an additional PlainTime parameter. Both of them must be converted into a *Temporal.PlainTime* object. However, for the second argument, the *convertJavaToJavaScriptPlainTime* function must be used. Additionally, a optional parameter can be defined, which defines the rounding mode that can be used. After calculating the difference of both PlainTime object, a *JSTemporalDuration* object will be returned.
- **Temporal.PlainTime.prototype.since**
  The since function is implemented in the class *JSTemporalPlainTimeSince*, and works identically to the previously mentioned function. It calculates the difference between both PlainTime object and returns a *JSTemporalDuration* object.

- **Temporal.PlainTime.prototype.round**
  The round function is implemented in the class *JSTemporalPlainTimeRound*. It has a PlainTime receiver object, which needs to be converted into a *Temporal.PlainTime* object, as well as a parameter called *roundToParam*, which defines the unit, to which the PlainTime object should be rounded to. This function uses the half expand rounding mode, and returns a new *JSTemporalPlainTime* object.

- **Temporal.PlainTime.prototype.equals**
  The equals function is implemented in the class *JSTemporalPlainTimeEquals*. The function is split into two seperate functions. One that expects that the second parameter is already a *Temporal.PlainTime* object, and the other, that does expect it not to be. Both methods have a PlainTime receiver object, as well as an additional PlainTime parameter. For the first method, only the receiver object must be converted. For the seond, both of them must be converted into a *Temporal.PlainTime* object. However, for the second argument, the *convertJavaToJavaScriptPlainTime* function must be used. Both function return a boolean value, which is either true or false.

- **Temporal.PlainTime.prototype.toString**
  The toString function is implemented in the class *JSTemporalPlainTimeToString*, and converts a PlainTime object into a wall-clock time.

- **Temporal.PlainTime.prototype.toLocaleString**
  The toLocaleString function, which is implemented in the *JSTemporalPlainTimeToLocaleString* class works identically to the previous function. It converts the parameter *thisObj* and returns a wall.clock time.

- **Temporal.PlainTime.prototype.toJSON**
  The toJSON function is implemented in the class *JSTemporalPlainTimeToJSON*, however, it is the same as the toString function, and should like for the other classes not be called directly.

- **Temporal.PlainTime.prototype.valueOf**
  The function valueOf is implemented in the class *JSTemporalTimeZoneValueOf*. This function only throws a exception if it is called, because, by default, valueOf should not be supported by the *Temporal.TimeZone* class.

- **Temporal.PlainTime.prototype.toZonedDateTime**
  The toZonedDateTime function is implemented in the class *JSTemporalPlainTimeToZonedDateTime*. It has a PlainTime receiver object, which needs to be converted into a *Temporal.PlainTime* object, as well as a parameter called *item*. This parameter holds a *Temporal.PlainDate* object, as well as a *Temporal.TimeZone* object from the IANA time zone database. The function returns a *JSTemporalZonedDateTime* object.

- **Temporal.PlainTime.prototype.toPlainDateTime**
  The toPlainDateTime function is implemented in the class *JSTemporalPlainTimeToPlainDateTime*. It has a PlainTime receiver object, which needs to be converted into a *Temporal.PlainTime* object, as well as a parameter which represents a *Temporal.PlainDate* object. The function returns a *JSTemporalPlainDateTime* object.

- **Temporal.PlainTime.prototype.getISOFields**
  The getISOFields function is implemented in the class *JSTemporalPlainTimeGetISOFields*. It only has a PlainTime receiver object, which must be converted into a *Temporal.PlainTime* object. This function returns all units defined in the PlainTime object.

4.3.5 Temporal PlainDate

In this section we are going into more detail about the methods implemented in *Temporal.PlainDate* class. Those methods include:

- **Temporal.PlainDate.prototype.with**
  The function with is implemented in the class *JSTemporalPlainDateWith*. It has a receiver PlainDate object, which must be converted inside the function, as well an additional PlainDate parameter, which is used to set specific units of the first parameter to the defined values. Similarly to the with function mentioned for PlainTime, the PlainDate object *"2000-01-24"* with the day changed to 30, the function would return a new PlainDate object, with the value *2000-01-30*.

- **Temporal.PlainDate.prototype.withCalendar**
  The function withCalendar is implemented in the class *JSTemporalPlainDateWithCalendar*. It has a receiver PlainDate object, which must be converted inside the function, as well as a *Temporal.Calendar* object. The function returns a new *JSTemporalPlainDate* object, which is projected into the defined Calendar object.

- **Temporal.PlainDate.prototype.add**
  The function add is implemented in the class *JSTemporalPlainDateAdd*, and adds a duration to a the receiver PlainDate object, which must be converted inside the function. A new *JSTemporalPlainDate* object is returned at the end of the function.

- **Temporal.PlainDate.prototype.subtract**
  The function subtract is implemented in the class *JSTemporalPlainDateSubtract*. It works in a similar fashion as the previous mentioned function. But instead of simply adding a duration to a PlainDate object, it first changes the sign of the duration, because adding it. The result is yet again a new *JSTemporalPlainDate* object.

- **Temporal.PlainDate.prototype.until**
  The until function is implemented in the class *JSTemporalPlainDateUntil*. It has a PlainDate receiver object, as well as an additional PlainDate parameter. Both of them must be converted into a *Temporal.PlainDate* object. However, for the second argument, the *convertJavaToJavaScriptPlainDate* function must be used. Additionally, a optional parameter can be defined, which defines the rounding mode that can be used. After calculating the difference of both PlainDate object, a *JSTemporalDuration* object will be returned.

- **Temporal.PlainDate.prototype.since**
  The since function is implemented in the class *JSTemporalPlainDateSince*, and works identically to the previously mentioned function. It calculates the difference between both PlainDate object and returns a *JSTemporalDuration* object.

- **Temporal.PlainDate.prototype.equals**
  The equals function is implemented in the class *JSTemporalPlainDateEquals*. It has a PlainDate receiver object, as well as an additional PlainDate parameter. Both of them must be converted into a *Temporal.PlainDate* object. However, for the second argument, the *convertJavaToJavaScriptPlainDate* function must be used. The function returns a boolean value, which is either true or false.

- **Temporal.PlainDate.prototype.toString**
  The toString function is implemented in the class *JSTemporalPlainDateToString*, and converts a PlainDate object into a string that represents the date in *ISO-8601*.

- **Temporal.PlainDate.prototype.toLocaleString**
  The toLocaleString function, which is implemented in the *JSTemporalPlainDateToLocaleString* class works identically to the previous function. It converts the parameter *thisObj* and returns human-readable date as a string.

- **Temporal.PlainDate.prototype.toJSON**
  The toJSON function is implemented in the class *JSTemporalPlainDateToJSON*, however, it is the same as the toString function, and should like for the other classes not be called directly.
- **Temporal.PlainDate.prototype.valueOf**
  The function valueOf is implemented in the class *JSTemporalPlainDateValueOf*. This function only throws a exception if it is called, because, by default, valueOf should not be supported by the *Temporal.PlainDate* class.
- **Temporal.PlainDate.prototype.toZonedDateTime**
  The toZonedDateTime function is implemented in the class *JSTemporalPlainDateToZonedDateTimeNode*. It has a PlainDate receiver object, which needs to be converted into a *Temporal.PlainDate* object, as well as an parameter called *item*, which holds a *Temporal.PlainTime* object, as well as a *Temporal.TimeZone* object. The function returns a new *JSTemporalZonedDateTime* object, which combines the PlainDate object with the item parameter.
- **Temporal.PlainDate.prototype.toPlainDateTime**
  The toPlainDateTime function is implemented in the class *JSTemporalPlainDateToPlainDateTime*. It has a PlainDate receiver object, which needs to be converted into a *Temporal.PlainDate* object, as well as optional PlainTime parameter. The function returns a *JSTemporalPlainDateTime* object.
- **Temporal.PlainDate.prototype.toPlainYearMonth**
  The toPlainYearMonth function is implemented in the class *JSTemporalPlainDateToPlainYearMonth*. It has a PlainDate receiver object, which needs to be converted into a *Temporal.PlainDate* object. The function returns a object which represent year and month of the PlainDate object.
- **Temporal.PlainDate.prototype.toPlainMonthDay**
  The toPlainMonthDay function is implemented in the class *JSTemporalPlainDateTotoPlainMonthDay*. It has a PlainDate receiver object, which needs to be converted into a *Temporal.PlainDate* object. The function returns a object which represent month and day of the PlainDate object.
- **Temporal.PlainDate.prototype.getISOFields**
  The getISOFields function is implemented in the class *JSTemporalPlainDateGetISOFields*. It only has a PlainDate receiver object, which must be converted into a *Temporal.PlainDate* object. This function returns year, month, week and day of the PlainDate object.

4.4 ForeignObjectPrototypeNode

The *ForeignObjectPrototypeNode* class defines how the mapping between the foreign Java type and the native JavaScript type should be done.to the program how the mapping between the foreign Java type and the native JavaScript type should be done. For example, if the value is a foreign Instant, Temporal.Instant.prototype should be used.

Listing 4.11: The method that determines which Temporal class object is used

```
@Specialization(limit = "InteropLibraryLimit")
public JSDynamicObject doTruffleObject(Object truffleObjct,
                @CachedLibrary("truffleObject") InteropLibrary interop) {
    JSRealm realm = getRealm();
    ...
    else if (interop.isInstant(truffleObject)) {
        return realm.getTemporalInstantPrototype();
    }
    else if (interop.isDuration(truffleObject)) {
        return realm.getTemporalDurationPrototype();
    }
    else if (interop.isDate(truffleObject)) {
        return realm.getTemporalPlainDatePrototype();
    }
    else if (interop.isTime(truffleObject)) {
        return realm.getTemporalPlainTimePrototype();
    }
    else if (interop.isTimeZone(truffleObject)) {
        return realm.getTemporalTimeZonePrototype();
    }
    ...
    else {
        return realm.getObjectPrototype();
    }
}
```

Chapter 5

Testing

In the final chapter of this thesis we are going to look at some tests that are written in order to confirm that every function is working as expected. This chapter describes how tests are implemented, what is tested or omitted from testing, and why several tests currently fail.

The tests have been written in the class *TemporalInteropTest*, which was created for the purpose of writing the tests for every *Temporal class* conversion methods. The tests were written in Java, using the *JUnit* framework. However, different testing methods could also be used.

For the most part, one test has been implemented for each *Temporal* class method. It is not necessary to implement multiple tests, because the task of this project was not to test, if the methods work for each scenario, but test if the conversion was done correctly. That usually only requires one test. For the most part, every test is implemented in a similar fashion, meaning that test could be copied and pasted the same way, for each method, only changing a couple of lines of code.

5.1 Tests

In the Listing 5.1, you can observe, how such an test can be implemented. To mark a method as as test, it must have the annotation *@Test*. The tests do not need any *@Before* or *@After* methods, but instead constructs and deconstructs everything they need by themselves.

Every test must define a String code, which defines the code that should be called as *JavaScript*. As can be observed in Listing 5.1, line 3, this String object includes a variable called javaInst, implying that this variable is a Java Instant object. This object is defined in line 5, and must be a *java.time* object. In this case, it is a Instant object, which holds the number *100_000_000*. In order to use the instant object, a Context object must be created. Line 7-11, showcase how such a context has been created. Basically, it defines the language which should be used (line 7), configures which public constructors, methods, fields or classes are accessible by the user (Line 8). Typically, this should be set to the value *HostAccess.ALL*, which allows a full unrestricted access of the host objects. The parameter in line 9, called *allowExperimentalOptions* should be set to the value true. Additionally, a context can define a couple of options, which can be defined as shown in line 10-11.

Now that the context has been created, a value must be bound to it. That is done by defining an identifier, which must be of the same name, that is used in Listing 5.1, line 3, as well as the *Temporal* class object, defined in line 5. The instant value should now be bound to the context, and can be used to evaluate the code defined in line 5. This value will return a Value class object, as shown in line 13, which holds the return value of any *Temporal* class method that was used. Now that we have a result, we can use the method *assertEquals()* to test, if the the return value is the expected result.

Listing 5.1: A JUnit test

```
@Test
public void testInstantAdd() {
    String code = "javaInst.add(Temporal.Duration.from('PT1H)).
    epochMilliseconds;";

    java.time.Instant inst = Instant.ofEpochMilli(100_000_000);

    try (Context ctx = context.newBuilder("js").
            allowHostAccess(HostAccess.ALL).
            allowExperimentalOptions(true).
            option(JSContextOptions.FOREIGN_OBJECT_PROTOTYPE_NAME, "true").
            option(JSContextOptions.TEMPORAL_NAME, "true").build()) {
        ctx.getBindings(ID).putMember("javaInst", inst);
        Value result = ctx.eval(ID, code);
        Assert.assertEquals(100_000_000L, result.asLong());
    }
}
```

5.2 Problems

As mentioned at the start of this chapter, some methods do not return the result that is expected of them, or upright throw an exception. Unfortunately, the things that need to be changed in order to make these tests work can not be changed that easily. They have to do with the implementation of GraalVM, and have to be solved in future work. Usually, these tests are ignored using the annotation *@Ignore*. Let us look at some of these problems.

5.2.1 Arity Error

An *ArityException* is thrown, when a executable or instantiable object was provided with the wrong number of arguments. That error is not the fault of how the test was written. The problem is, that the *JavaScript* method, that is used in the test has the same name as a *Java* method. For example, the *Java* method *until* has two arguments. However, the *JavaScript* does only have one argument, and because the compiler does first check for Java functions, an *ArityException* is thrown, saying, that an argument is missing.

5.2.2 Type Error

The *Type Error* is called, when a *Temporal* class object is expected, but no such object has been set. This error happens, quite similarly to the *ArityException*, because the function does exist in both languages. However, this time, the number of arguments was correct, and the *Java* method was used. However, in doing so, the return value is wrong, and a error message like *TypeError: Temporal.Duration expected* is written into the terminal. An example for this would be the *negated()* function.

5.2.3 Other problems

Another error does occur among the tests. That error only appears for constructor methods, and it will be left for future work to resolve that specific issue.

Chapter 6

Conclusion

In this project, support for using java.time.* in JavaScript Temporal code has been implemented. A large number of use-cases can be covered by the solution provided. Some areas remain unsolved for the moment and require further consideration. Those unsolved issues include the Temporal.Calendar, PlainMonthDay, PlainYearMonth, as well as the ZonedDateTime class objects. At this point, a differentiation between these class objects, and the class objects that have been implemented in this thesis is not possible, and remains to be solved in future work.

In the extent of this thesis, all implementations have been tested. It should be noted, that those tests only determine if the conversion between Java Temporal to JavaScript Temporal is possible. Most functions have additional parameters, which have not been used in the tests. For example optional parameters that determine if the value should be rounded. Those tests remain for future work.

However, some tests failed regardless. As mentioned in Chapter 5, some challenges could not be resolved in this thesis. For reminder, Arity Errors, and Type Errors, which occurred, because there exists a conflict between the used methods and the Java.time methods. Therefore this challenge remains to be resolved by future work.

Bibliography

[1] [n. d.] ECMAScript Description. Retrieved 10/05/2022 from https://www.ecma-international.org/technical-committees/tc39.

[2] [n. d.] Graal.js. Retrieved 11/17/2002 from https://github.com/oracle/graaljs.

[3] [n. d.] GraalVM. Retrieved 11/17/2002 from https://www.graalvm.org/latest/docs/getting-started.

[4] [n. d.] ISO 8601. Retrieved 10/05/2022 from https://en.wikipedia.org/wiki/ISO_8601.

[5] [n. d.] Proposal-Temporal. Retrieved 10/05/2022 from https://github.com/tc39/proposal-temporal.

[6] [n. d.] Temporal Duration. Retrieved 01/24/2023 from https://tc39.es/proposal-temporal/docs/#Temporal-Duration.

[7] [n. d.] The correspondence between types and machine-readable strings. Retrieved 01/15/2023 from https://tc39.es/proposal-temporal/docs/.

[8] [n. d.] Truffle. Retrieved 11/17/2002 from https://www.graalvm.org/22.0/reference-manual/java-on-truffle.

[9] [n. d.] Truffle-based AST. Retrieved 10/05/2022 from https://www.javascripttutorials.net/graal-js-javascript-on-the-jvm.

[10] [n. d.] Unix Time. Retrieved 10/05/2022 from https://en.wikipedia.org/wiki/Unix_time.