Value Types in Typescript for JValue

BACHELOR THESIS

Mert Baran

Submitted on 22 October 2021

Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik
Professur für Open-Source-Software

Supervisor:
Prof. Dr. Dirk Riehle, M.B.A.

Versicherung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, 22 October 2021

License

Erlangen, 22 October 2021

Abstract

Over the past years, TypeScript has increasingly been gaining popularity due to its nature of providing functionalities to ease the development of scalable and robust applications whilst syntactically being a superset of JavaScript. With the growing complexity of data-driven environments, it is essential for programming languages to cope with value types beyond their primitive data types to capture the semantics of intangible data, such as systems of measurement, thus increasing readability and solidity across the codebase. By creating a test-driven framework in TypeScript, this thesis lays out different methods to efficiently implement value types, discusses their benefits as well as drawbacks, and ensures the reliability of the framework by integrating it into an existing data-driven service.

Contents

1 Introduction

Objects and Values are fundamentally different concepts. (Bäumer et al., 1998) Values are intangible data that have meaning in their domain. Conceptually, they are identity-less and live beyond the boundaries of time. Thus, two expressions of values are equal when they semantically represent the same meaning in their domain (Riehle, 2006). Contrarily, objects are often not characterized by their semantics; they often are defined 'by a thread of continuity and identity' (Evans, 2014). They usually express entities that can be created and deleted when needed. Consequently, they often describe tangible, meaning existing and feasible, components of the real world.

Values, when reasonably used, enrich the design of software. They help us understand complex real-world data by breaking information down to their logical represetations. Furthermore, depending on their implementation, they can enforce the use of an immutable programming paradigm, which is characterized by its read-only behavior on values. (Riehle, 2006)

Despite the many benefits that value types bring with them and the mental conception of them being around since the dawn of time, many programming languages lack the concept for creating custom values beyond their primitive ones. This, however, can be overcome with the use of design patterns that allow the creation and definition of new value types on top of a language's existing value types. (Bäumer et al., 1998)

One of the languages that fall into that category is TypeScript. Despite providing a magnitude of functionalities and tools that are built on top of JavaScript, the language still lacks support for creating custom values. The absence of such a concept in TypeScript can be especially blatant since one of JavaScript's forte's lies in creating data-driven applications and APIs.

From the given occasion, this thesis implements an object-oriented framework that allows the creation of custom value types on top of TypeScript's existing value types. Key functionalities are applying predefined constraints to values but also creating custom ones. Transformer functions allow for the conversion between two types given a transformation strategy. Furthermore, inferring the

type of custom value types and verifying concrete values are essential capabilities.

Chapter 2 discusses the differences between value types and object types and gives a brief introduction to TypeScript. Chapter 3 defines the objectives of the framework and encompasses the requirements that should be met and evaluated with the use of later in Chapter 6. Chapter 4 lines out the framework's architecture, followed by the implementation in Chapter 5.

2 Problem Identification

2.1 Value Types and Object Types

Generally, when modeling a software architecture, it is common to implement real-world abstractions as Entities '(a.k.a. Reference Objects)' (Evans, 2014). Entities are objects that have distinct identifiers. They often are distinguished by a single property of the object. Other properties, generally referred to as attributes, further describe the object at hand. Since they serve no classification purpose, they can be changed throughout the Objects lifecycle.

```
class User {
    constructor(private readonly id: number,
                private name: string) {}
    setName(name: string) {
        this.name = name;
    }
    equals(other: User): boolean {
        return other.id == this.id
    }
}
let user1 = new User(1, "John")
let user2 = new User(2, "John")
// false, not the same id
console.log(user1.equals(user2))
```

Code 2.1: A common use case for entities is creating models for database schemas. The class on hand serves as a model for a User table schema. The equality between two instances solely is determined by their identifier.

Instances of Object types normally have a lifecycle. After creating them, they will eventually be deleted if the reference is no longer needed. It is possible that when a component deletes an instance, that another entity still could have a reference to it. This, and the fact that instances of object types are generally

```
type MoneyType = { readonly amount: number,
                   readonly currency: "USD" | "EUR" }

let a: MoneyType = { amount: 10, currency: "USD" }
let b: MoneyType = { amount: 10, currency: "USD" }
// true
console.log(JSON.stringify(a) == JSON.stringify(b))
```

Figure 2.1: The equality between value types is determined by their structural equality.

mutable, can make developing software, especially in larger-scale systems, very error-prone. (Bäumer et al., 1998)

Value types in programming languages generally encompass only primitive types, such as numbers or strings. However, when working in a domain-driven environment, it can be of crucial help to capture the semantics of data. Not only does it allow to tackle down the complexity of the codebase by grouping cohesive to simple descriptive representations, but it also can make the development process less error-prone due to dealing with immutable values. The value object pattern aims to achieve this by creating domain-specific value types, which behave like built-in values.

Value objects, or values, are instances of value types. The latter defines a schema and criteria that their instances need to satisfy. When implementing this pattern, habitually, the schema is built on top of the language's existing primitive types.

Value Objects, unlike entities, don't have unique identifiers. Their identity only is determined by the values that their properties own so that two instances are equal when their internal values are structurally the same.(Bäumer et al., 1998) Values also have no lifecycle. Even though their concrete instances can be created or deleted, conceptually, they don't exist in time; they're intangible and given. Since values don't have identifiers, hence it's easy to serialize them since there's no need to implement a hash method to get unique identifiers for the objects.

2.1.1 Benefits and drawbacks of Value Types

Value Types and values can help to break down complexity across the codebase. They can help prevent errors and give the programmer a better understanding of the code. That is a result of the mental shift from changing the state of an object and thus having to validate a new internal state across different components to not modifying the state but returning a new value instance that already has to exist according to the schema provided by the ValueType.

Other benefits of typing custom values are the ability to port the defined value

types to other domains, to be able to serialize values simply by recursively serializing their internal structure, and that they're not bound to a lifecycle and are interchangeable, thus not having to keep track of references to them. (Riehle, 2006)

The main drawback of the Value Object pattern is the performance penalty, which ensues by creating heavyweight value types. Named value types, such as postal addresses (2006), cause a proliferation of values that are mostly unique and share little to no properties with other values of their type. The use of the Value Object pattern could also lead to 'more complicated code'(Riehle, 2006), which is a result of the read-only programming paradigm, that a developer, first, would have to get accustomed to. Generally, it is advised to use the Value Object pattern if the resulting values are lightweight and if the concept at hand represents a value. (2006)

2.2 The TypeScript language

The programming language JavaScript was first released more than two decades ago. Initially, the purpose of the language was to allow the dynamic manipulation of static web pages. Since then, JavaScript has evolved rapidly and is now the most popular programming language on GitHub[1]. That is partly due to the versatility of the language. JavaScript finds its use-cases in creating dynamic web pages, creating server-side applications with node.js, building Desktop apps with Electron, and so forth.

However with JavaScript's dynamic typing system, developing large-scale applications can be very complex and error-prone. To combat this, TypeScript, a syntactic superset of JavaScript, was introduced by Microsoft in 2012.

TypeScript allows developers to add type annotations to their code, thus making the codebase more readable, scalable and maintainable. With its roaring open-source community and popular JavaScript frameworks, such as AngularJS, migrating their codebase to to TypeScript, it has gained enormous popularity within the last few years.

As of the current version of TypeScript[2], it, like other languages, does not provide a built-in concept to create custom value types. Nonetheless, it gives a solid basis to create a value type framework.

[1] https://madnight.github.io/githut/
[2] https://www.typescriptlang.org/docs/handbook/release-notes/typescript-4-4.html

3 Objective Definition

3.1 A framework for ValueTypes

The primary goal of this thesis is to design an object-oriented framework for the purpose of providing custom value types.

A framework is an 'object-oriented' design that usually 'covers one particular technical domain' and 'can be reused' (Riehle, 2021). Since the concept of value types is relatively abstract and they can be used in many domains, it makes sense to build a framework for creating and validating such types.

3.2 Requirements analysis

The majority of the items on the following list were composed before starting working on the framework. They create an outline of the desired outcome and will be used to evaluate the framework in chapter 6.

1. **The framework should be extensively tested**It's highly important to test every little aspect of the framework. After all, it is designed to deal with complex values that could potentially be falsely manipulated and therefore become unusable.
2. **Providing an interface so that custom ValueTypes can be defined** Since the purpose of this framework is to allow potential developers to define their custom Value Types, this framework must provide easy and understandable methods to do so. The framework should limit the primitive types one can use to create a schema. This is because, when using some types, such as the top types, any and unknown, as base types in the framework, it could lead to loosely typed schemas.

```
let personType = new ObjectType({
    firstName: new StringType(),
    middleName: new StringType().makeOptional()
    lastName: new StringType()
```

```
})
```

3. **The ability to infer static types from value type instances** It should be possible to retrieve the static type of a value type instance, which would help immensely when validating value objects. The inferred type for the example above should be:

```
let a: inferValueType<typeof personType>;
typeof a === {
    firstName: string,
    middleName?: string,
    lastName: string
}
```

4. **Constraints to be used on Value Types** It should be possible to add constraints on value types to restrict the range of values a value type can take. There should be multiple predefined rules that a developer could use on predefined value types.

```
let personType = new ObjectType({
    firstName: string.withMinLengthRestriction(2),
    middleName?: string,
    lastName: string
}
```

5. **Implementing distributable domain packages** One of the many benefits to value types is that once they're defined, they can be reused. The framework should allow for a developer to register custom-created value types, rules, and transformations.
6. **Implementation of a reusable domain package** With the previously defined domain-registration mechanism, the provided framework should at least have the following domain with its value types implemented: QuantityUnitType

3.3 Reference Projects

3.3.1 jvalue/value-objects

The jvalue/value-objects[1] project is a framework that implements Value Types in Java and was the main source of reference for this framework.

[1] https://github.com/jvalue/value-objects

3.3.2 jvalue/kernel

My first attempt to create a value type framework in TypeScript was with the heavy usage of TypeScript's compiler API to implement a reflection API and optimized evaluation characteristics for ValueTypes. This, however, resulted in unstable and unpredictable behavior so that it never went past the experimental phase. With pointers on how to parse value objects from the jvalue/kernel[2] team, the current version of the framework doesn't rely on the compiler API.

[2]TBR.

4 Architecture

4.1 Using the Meta-Object Protocol

Since TypeScript's objects generally provide little to no meta-information or inherited serialization or equality methods, it is best to make use of the Meta-Object Protocol design pattern for the framework.

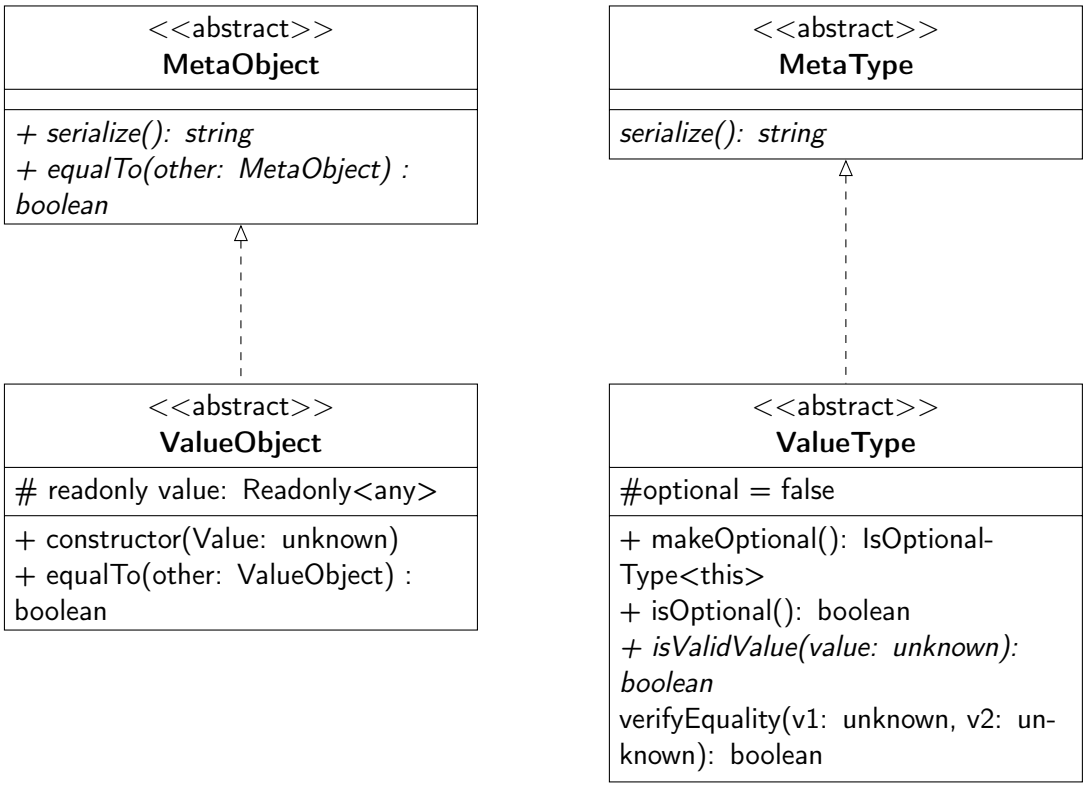
The Meta-Object Protocol provides an internal typing system in the framework on top of TypeScripts' existing type system. As a result, classes that inherit the MetaType or the MetaObject class share consistent functionality between them, which is very useful when it comes to abstract frameworks and produces unified behavior among the inheriting subclasses.

One of the functionalities that every value object should provide is to check if they're equal to another value object. Furthermore, subclasses should provide their serialization strategy.

```
export abstract class MetaObject {
    abstract serialize(): String;
    abstract equalTo(other: MetaObject): boolean;
}
```

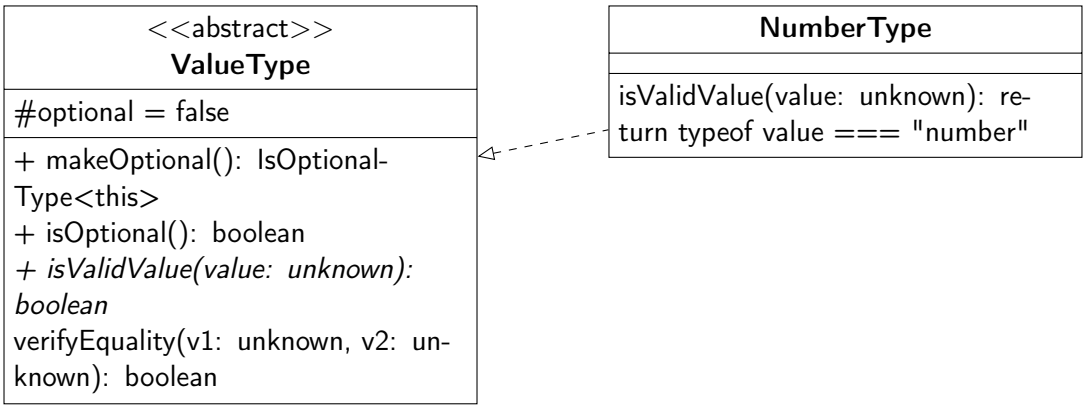
Figure 4.1: The MetaObject enforces subclasses to implement the serialize and equalTo methods.

Going a level down from the MetaObject class, the ValueObject defines basic functionality,



The Value Object class is an abstract class that extends the MetaObject and should handle equality checks to other value objects. It may or may not hold a reference to its type. The reference to its value type is not necessary. A value can theoretically hold an infinite amount of representations, just like integers in the real world. It is the type, that determines, whether or not a value is valid in its schema.

The Value Type, on the other hand, is the core element of this framework. It is abstract by design, so that concrete value types can create instances by extending the class. All of the framework's logic regarding, whether a value is valid or not or whether or not a rule is applicable on a type should be handled through this specific class. Subclasses extending this class should only have to set a few abstract members.



The framework should provide typical types[1] as core value types, which can then be used to implement more complex value types such as objects or union types on top of them. Rules should be chainable to the Value Types, so that custom and modified value types can be instanciated fairly easily.

To allow the creation of custom value schemas, it is necessary to define the base types first. These include: With these types, it is possible to build safe and

```
type TypicalTypes = number | string | boolean | bigint
                          | TypicalType<TypicalTypes>[];
```

reliable schemas. Top types like any or unknown or bottom types are too loosely or respectively too tightly bound so that the behavior of otherwise well-typed schemas could be undetermined.

Furthermore, it suggests itself creating a container for all the typical types listed. That is because these base types can hold their own rules and have their validation mechanism.

```
export class BooleanType extends TypicalType<boolean> {
    /** Own logic and validation */
}
```

When trying to implement constraints on the previously designed value types, one could go about different ways. An obvious way to implement rules on value types is to have the rules defined directly in the extended value types class.

The benefit of this implementation would be, that rules are easily chainable and properties could easily be accessible by the linter. A disadvantage is the pollution of the class itself by many restrictions and properties. Another disadvantage of that method is that rules, which can be applied to different value types have to be redefined in each value type again.

[1] https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

```
export class NumberType extends TypicalType<number> {
    // ...
    public withLowerBoundRestriction
    (l: number, i: boolean = false) : NumberType {
        this.lowerBound = i ? l - 1 : l
        this.validationChainAdd((value: number) => number < this.lowerBound)
        return this
    }
}


export class StringType extends TypicalType<string> {
    // ...
    public minimumLength(l:number) : StringType {
        this.minLength = l;
        this.validationChainAdd((value: string)
        => value.length > this.minLength) ----- repeating
        return this;
    }
}
export class ArrayType<T extends TypicalTypes>
    extends TypicalType<TypicalType<T>[]> {
    // ...
    public minimumLength(l:number) : StringType {
        this.minLength = l;
        this.validationChainAdd((value: string)
        => value.length > this.minLength)   ----- repeating
        return this;
    }
}
```

Since this framework aims to provide methods that let users declare their own rule, this approach won't suffice. Next, it is plausible to divide value types and rule classes from each other and provide an interface that lets value types add rules to their instances. The common approach to do so would be like this:

```
interface IRule {
    evaluate(value: unknown) : boolean;
}
----
class MinLengthRule<T extends string|TypicalTypes[]> implements IRule {
    constructor(protected readonly minLength: number) {
        super()
```

```
    }
    evaluate(value: T): boolean {
        return value.length >= this.minLength;
    }
}
----
abstract class ValueType {
    protected validation: (value: unknown) => boolean[] = []
    withRule(r: IRule) {
        validation.push(r)
        return this;
    }
}
```

This approach allows users to define their own rules. The greatest shortcoming of it, however, is that one no longer can apply rules to a value type by accessing it directly by it's member.

5 Implementation

The implementation of the value types framework was pretty straightforward. The first step was to implement the base types. A generic TypicalTypes container holds all primitive types. This has the advantage that, when trying to infer the type from a value object, it's easier to infer the base types extending the container themselves.

Here, with the static type schemaType<u> it is easily possible to create a type for an object by simply inferring the value type itself. That is especially useful, when testing the framework and statically creating new values.

```
type schemaType<u> = u extends ObjectType<infer y> ? schemaTypeHelper<y>  : never

type getTypeHelper<A> = A extends TypicalType<infer l> ? l : never

type schemaTypeHelper<T extends {}> = {
    [K in keyof T]: T[K] extends
     IsOptionalType<TypicalType<infer u>> ? Partial<u> :
                    T[K] extends
    UnionType<infer u> ? getTypeHelper<u[number]> :
                    T[K] extends
    TypicalType<infer u> ? u :
                    T[K] extends
    ObjectType<infer y> ? schemaTypeHelper<y> : never;
        }
```

5.1 Implementation of the QuantityType class

Quantities are complex representations of intangible data. Hence, it suggests itself modeling it as a value type. A QuantityType describes how different units correlate with each other. For instance, when dividing distance with time, we get a new composed unit of speed. To implement Quantity Type, first, it was

mandatory to create the SIUnit value object and respectively the SIUnit value type.

Since SIUnits consist of different units, it is of help to first create a static supply of base units.

```
public static readonly None: Units = { l: 0, w: 0, t: 0,
 a: 0, te: 0, s: 0, lu: 0 }

public static readonly l: SiUnit = new SiUnit({ l: 1 })
public static readonly t: SiUnit = new SiUnit({ t: 1 })
public static readonly w: SiUnit = new SiUnit({ w: 1 })
public static readonly a: SiUnit = new SiUnit({ a: 1 })
public static readonly te: SiUnit = new SiUnit({ te: 1 })
public static readonly s: SiUnit = new SiUnit({ s: 1 })
public static readonly lu: SiUnit = new SiUnit({ lu: 1 })
```

To perform arithmetic operations on these units, it, according to the core concept of value objects, must a new SIUnit.

```
multiply(other: SiUnit): SiUnit {
    let tmp = { ...SiUnit.None };
    Object.getOwnPropertyNames(tmp).forEach(key => {
        tmp[key as keyof typeof tmp] = this.getValue()[key as keyof typeof tmp]
            + other.getValue()[key as keyof typeof tmp]
    });
    return new SiUnit(tmp as Units);
}
```

6 Evaluation

1. **The framework should be extensively tested** The framework was thoroughly tested with the Jest testing suite.
2. **Providing an interface so that custom ValueTypes can be defined** Users of the interface can use already defined value types or they can create new ones if they like.
3. **The ability to infer static types from value type instances** It is possible to infer static types from type schemas.
4. **Constraints to be used on Value Types** Values can be restricted with rules.
5. **Implementing distributable domain packages** While a user can add their own rules and it to their custom domain, it is not possible to create custom domain packages as of now
6. **Implementation of a reusable domain package** The

7 Conclusion

Value Types help make codebases more readable while also reducing complexity. They should be used, when dealing with complex environments that have domain-specific data in the foreground. While modeling every single component in the codebase would be an anti-pattern, one should consider the benefits of having a more universal language if the resulting objects are not too heavyweight. While it is very much possible to implement the value objects to programming languages that do not provide any concepts to create value types, a naive implementation could bring enhancements, such as compiler optimizations.

Appendices

References

Bäumer, D., Riehle, D., Siberski, W., Lilienthal, C., Megert, D., Sylla, K.-H. & Züllighoven, H. (1998). Values in object systems. *Techn. Ber. Zurich, Switzerland: UBS AG*.

Evans, E. (2014). *Domain-driven design reference: Definitions and pattern summaries*. Dog Ear Publishing.

Riehle, D. (2006). Value object. *Proceedings of the 2006 conference on Pattern languages of programs*, 1–6.

Riehle, D. (2021). *Adap c11 - object-oriented frameworks* [Accessed: 2021–10-07].