Value Objects

Dirk Riehle, FAU Erlangen

ADAP B06

Agenda

1. Values vs. objects
2. Value equality
3. Immutable objects
4. Shared value objects
5. The QuantityUnit type
6. Value type constructors
7. Value types in practice

1. Values vs. Objects

Values

Values are timeless abstractions; they have

- No life-cycle, no birth or death, and do not change
    - Unless you consider human invention of a value its birth
- No identity, cannot be counted, there is only “one copy”

Values are instances of value types (a.k.a. data types)

Objects

Objects are virtual physical entities in the real / modeled world; they

- Exist in time and have a life-cycle, i.e. they
  - Can be created, changed, shared, deleted
- Have an identity independent of their attribute values

Objects are instances of object types (a.k.a. classes)

Value vs. Object Semantics in Programming

Value semantics implies that

- Values are copied (unless made immutable)

Object semantics implies that

- Objects are moved around by reference (a value)

Examples of Value Types

So-called primitive or built-in [1] or atomic value types include

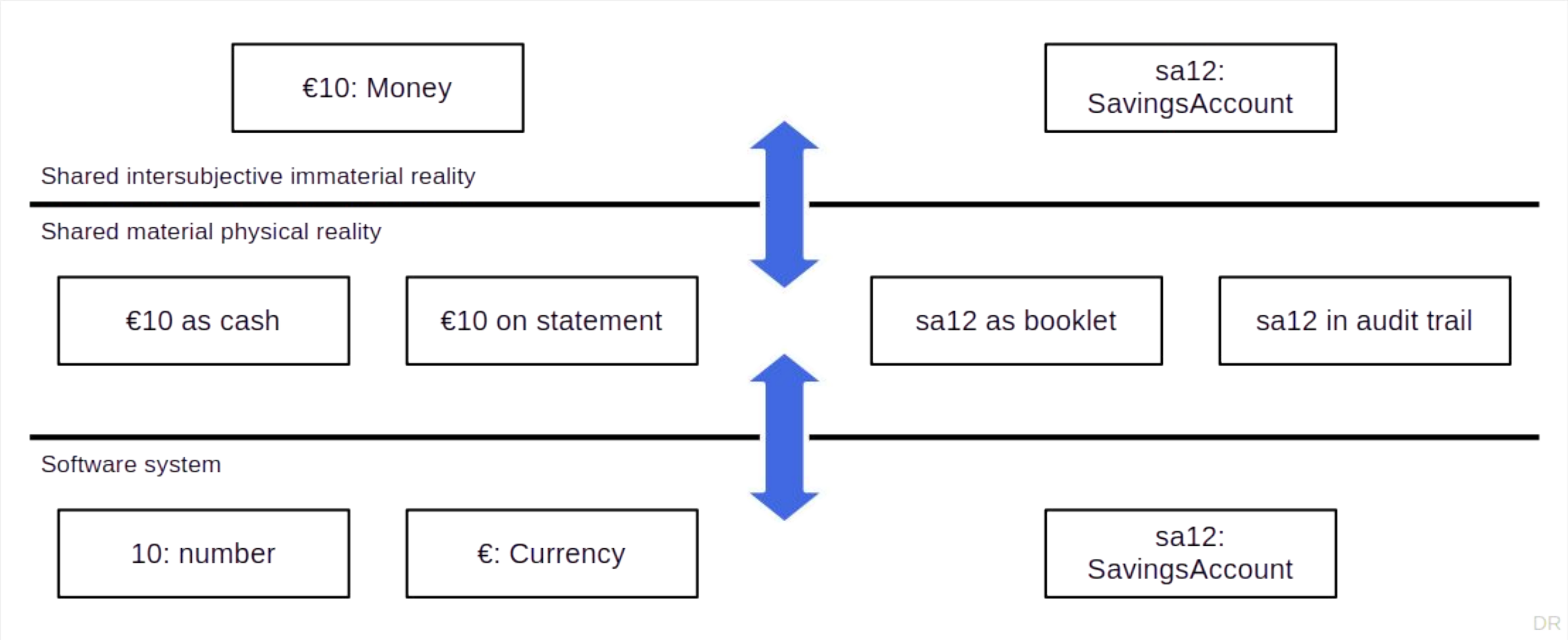
- boolean and bits
- numbers (int, long, float)
- characters and strings
- object references

Domain-specific value types, include, but are not limited to

- Coordinates and homogeneous names
- SI units and their ranges and restrictions
- Monetary amount, interest rate, stock symbol
- URLs, http return codes

[1] Rule of thumb: If it fits into a (8, 16, 32, …) register, it is a built-in value type

Occurrences and Representations of Objects and Values



Values and Objects in Programming Languages

There typically is no first-class concept of “value” [1]

Hence, values are often implemented as **immutable** objects

Mutable objects lead to side-effects, i.e. aliasing, a common source of bugs

[1] See https://dirkriehle.com/publications/1998-selected/values-in-object-systems/ for various techniques and approaches, some of which are discussed in this lecture

Advantages of Value Semantics

Using domain-specific value types

- Brings your programming closer to the problem domain
- Removes or restrain a major source of bugs (aliasing)
- May enhance system performance (shared, immutable)

Lack of identity allows for free copying

- No need for a separate database table to store values
- Value objects can be serialized in-line for network transfer
- No need for cross-process references in distributed systems

Object References (Identifiers)

Object identifiers are always values

- In-memory object pointer
- Handle (special type of pointer)
- External object identifier
- Primary keys

Object identifiers are not locations

- But locations are often (wrongly) used to identify objects

2. Value Equality

The Java equals() Contract for Any Object [1]

Applies to any runtime object (whether a regular object or a value object)

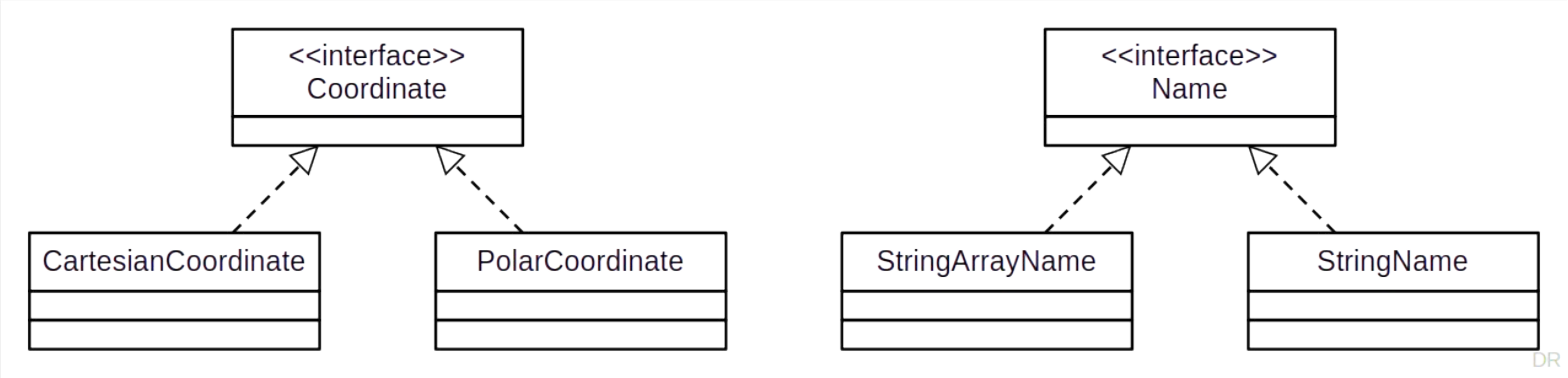
1. Reflexive: For any non-null reference x, x.equals(x) should return true.
2. Symmetric: For any non-null references x and y, x.equals(y) should return true if and only if y.equals(x) returns true.
3. Transitive: For any non-null references x, y, and z, if x.equals(y) returns true and y.equals(z) returns true, then x.equals(z) should return true.
4. Consistent: For any non-null references x and y, multiple invocations of x.equals(y) consistently return true or consistently return false, provided no information used in equals() comparisons on the objects is modified.
5. Null-Object: For any reference x, x.equals(null) should return false.

[1] See https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#equals-java.lang.Object-

The Equality Contract for Value Objects

Like the equality contract for any object plus

- An interface represents the value type
- The implementation classes represent an equivalency set



AbstractCoordinate.isEqual() Implementation

```
export abstract class AbstractCoordinate implements Coordinate {

    public isEqual(other: Coordinate): boolean {
        return (this.doGetX() == other.getX()) && (this.doGetY() == other.getY());
    }

    ...

}
```

3. Immutable Objects

Immutable Objects

Immutable objects are objects that

- Have no mutation methods (never change their state)

Immutable Value Objects

Value types are often implemented as immutable object classes

- (Former) mutation methods return another value object with the desired state

The result object may or may not be a new value object

```
export class CartesianCoordinate extends AbstractCoordinate {

    ...

    protected doSetX(x: number): Coordinate {
        return new CartesianCoordinate(x, this.y);
    }

    protected doSetY(y: number): Coordinate {
        return new CartesianCoordinate(this.x, y);
    }

    ...

}
```

Advantages and Disadvantages of Immutable Objects

Advantages

- No side effects from aliasing!
- Safe and perform well in concurrent programming

Disadvantages

- Increased object creation
- Your garbage collector may run hot quickly

Design by Contract and Immutable Value Objects

Preconditions

- Don’t change

Class invariants

- Are always the same: State did not change

Postconditions

- Move to the result object

4. Shared Value Objects

Shared Value Objects

One value → one object

Advantages and Disadvantages of Sharing Value Objects

Advantages

- Reduced (minimal!) memory consumption
- (Parts of) equality test can be reduced to identity test
- Hash code computation can be reduced to basic object hash code

Disadvantages

- Added programming complexity
- Performance penalty for organizing shared objects
- Gets more difficult with more than one implementation class

Handle / Body Idiom + Copy-on-Write [1]

The handle / body idiom

- Separates handle (reference) from body (payload)
- The handle is copied as the object is passed around

Copy-on-write

- Keeps the handle through which the write happens but
- Copies the body before mutation

This is a technique better suited than immutable objects for heavyweight objects

[1] Coplien, J. O. (1991). Advanced C++ programming styles and idioms. Addison-Wesley Longman.

5. The QuantityUnit Type

Design Exercise

Design a function that accepts a distance and a speed as the input

Compute and return the time it takes to go that distance at that speed

A Short Interlude About Requirements

Functional requirements missing

- Precision of calculation?
- What types of units?
- …

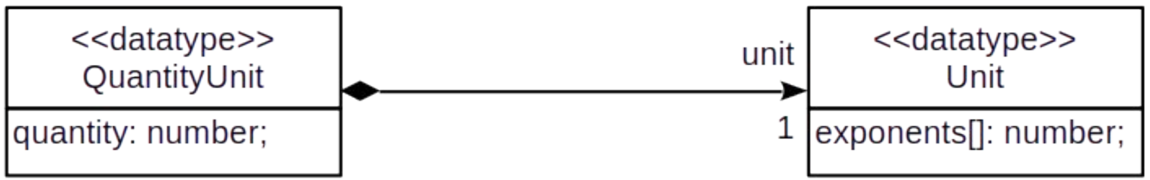
Non-functional requirements missing

- Speed of calculation?
- Concurrent computation?
- …

What's Wrong With This Solution?

```
export function calculateDuration1(distance: number, speed: number): number {
    return distance / speed;
}
```

Introducing QuantityUnit



Base Units of the Metric System

Quantity	Base Unit	Symbol
Length	meter	m
Mass	kilogram	kg
Time	second	s
Electric current	Ampere	A
Thermodynamic temperature	Kelvin	K
Luminous intensity	Candela	cd
Amount of substance	mole	mol

Implementing (SI) Units

```
export class SIUnit {
    protected exponents: number[] = [0, 0, 0, 0, 0, 0, 0];

    constructor(exponents: number[]) {
        this.exponents = [...exponents];
    }

    public isEqual(other: SIUnit): boolean {
       return this.exponents.every((v, i) => v === other.exponents[i]);
    }

    public add(other: SIUnit): SIUnit {
        IllegalArgumentException.assert(!other.isEqual(this));
        return new SIUnit(this.exponents);
    }
    ...

    public multiply(other: SIUnit): SIUnit {
        let result: number[] = [0, 0, 0, 0, 0, 0, 0];
        for (let i = 0; i < this.exponents.length; i++) {
            result[1] = this.exponents[i] + other.exponents[i];
        };
        return new SIUnit(result);
    }
    ...
}
```

A Better calculateDuration() Function

```
export function calculateDuration2(distance: number, speed: number): QuantityUnit {
    IllegalArgumentException.assertCondition(speed != 0);
    return new QuantityUnit(distance / speed, new SIUnit([0, 0, 1, 0, 0, 0, 0]));
}
```

Possible improvements

- Turn distance and speed into QuantityUnits
- Have base units ready as Unit instances e.g. for seconds

5. Value Type Constructors

Value Type Constructors

Arrays, i.e. [ ]

Enumerations, i.e. enum

Parameterized types a.k.a. generics i.e. <...>

Enumerations as Value Type Constructors

Enums provide shared values out of the box

- Immutability needs to be ensured by programming

Enums are great for documenting and type-checking codes!

Parameterized Types as Value Type Constructors

Common parameterized types are ranges and range restrictions [1]

[1] See https://github.com/jvalue/value-objects for a decent Java implementation

Parameterized Types as Value Type Constructors [1]

```
export class RangeBound<T> {

    protected value: T;
    protected inclusive: boolean;

    constructor(value: T, inclusive: boolean) {
        this.value = value;
        this.inclusive = inclusive;
    }

    public getValue(): T {
        return this.value;
    }

    public isInclusive(): boolean {
        return this.inclusive;
    }

}
```

```
export class Range<T> {

    protected lowerBound: RangeBound<T>;
    protected upperBound: RangeBound<T>;

    constructor(lb: RangeBound<T>, ub: RangeBound<T>) {
        this.lowerBound = lb;
        this.upperBound = ub;
    }

    public includes(value: T): boolean {
        let lowerValue = this.lowerBound.getValue();
        let upperValue = this.upperBound.getValue();
        ...
    }

    public getLowerBound(): RangeBound<T> {
        return this.lowerBound;
    }

    ...
}
```

[1] A simplified version of https://github.com/jvalue/value-objects

Example Data From the GeBOS System [1]

The GeBOS was a large C++ software to operate cooperative banks

It had about 50 unique base domain-specific value object classes

It had about 20 unique constructors, generating hundreds of value types

It had more than 200 enums representing various domain-specific codes

[1] Bäumer, D., Gryzcan, G., Knoll, R., Lilienthal, C., Riehle, D. & Züllighoven, H. (1997). Framework Development for Large Systems. Communications of the ACM, vol. 40, no. 10, pp. 52-59.

Summary

1. Values vs. objects
2. Value equality
3. Immutable objects
4. Shared value objects
5. The QuantityUnit type
6. Value type constructors

Thank you! Any questions?

dirk.riehle@fau.de – https://oss.cs.fau.de

dirk@riehle.org – https://dirkriehle.com – @dirkriehle

Legal Notices

License

- Licensed under the CC BY 4.0 International license

Copyright

- © 2012, 2018, 2024 Dirk Riehle, some rights reserved