WERCSTAT
ERP software developers

Value Types

wercstat.com

version 1.6, 2024-12-24

Steenovenweg 5
5708 HN Helmond

Tel 085 - 060 64 67
Mail info@wercstat.com
Web www.wercstat.com

BTW NL8101.85.106.B.01
KVK 17114283
IBAN NL33 RABO 0104 1128 59
BIC RABONL2U

Table of Contents

Project: com.wercstat.domain

Contact: info@wercstat.com

This document is part of the `wercstat` low-code framework (https://www.wercstat.com).

The following documents are available:

(1) `Wercstat Overview`: introduction to the framework

(2) `Wercstat Getting Started`: installation instructions and hello-world tutorial

(3) `Wercstat Value Types`: description of `Java` domain value types

(4) `Wercstat Server DSL`: description of the server-side Domain Specific Language

(5) `Wercstat Client DSL`: description of the client-side Domain Specific Language

---

Domain Value Types

Overview

Value Types are user-defined business types that wrap `Java` primitives. They improve source-code expressiveness, readability and type safety.

Most base domain types have overloaded constructors:

```
Amount amount1 = new Amount(10.0);
Amount amount2 = new Amount("10.0");
Amount amount3 = new Amount(BigDecimal.TEN);
```

An `Amount` is not the same as a `Price`, even though both are represented as `BigDecimal`. In a business context treating them as equal might indicate an error:

```
Amount amount = new Amount(10.0);
//Price price = amount; ①
```

① assigning an *amount* to a *price* makes no business sense

calculating an amount does make sense:

```
Quantity quantity = new Quantity(2);
Price price = new Price(3.15);
Amount amount = Amount.of(quantity, price); ①
```

① user defined method on value type *Amount*.

The same applies to String primitives:

```
Description description = new Description("Engine valve");
//DocumentCode documentCode = description; ①
```

① assigning a *description* to a *document-code* makes no business sense.

When assigning a `Price` to an `Amount`, or a `Reference` to a `Document`, it should be done explicitly, thereby acknowledging the distinct business concepts.

```
Amount amount = new Amount(10);
Price price = new Price(amount); ①
```

① throws an exception if scale and/or precision of *price* can not hold the *amount*

and

```
Description description = new Description("SLS01234");
DocumentCode documentCode = new DocumentCode(description); ①
```

① throws an exception if the *reference* value exceeds the length of a *document-code*.

Business Value Types increase type safety throughout the source code.

Compare:

```
SalesOrder createSalesOrder(①
    String documentCode,
    Customer customer,
    String reference,
    LocalDate orderDate,
    BigDecimal amount,
    BigDecimal discountPercentage){
    }
```

① mixed abstraction level; `Java` primitive `String` and entity-type `Customer`

to:

```
SalesOrder createSalesOrder(①
    DocumentCode documentCode,
    Customer customer,
    Reference reference,
    OrderDate orderDate,
    Amount amount,
    Percentage discount){
    }
```

① same abstraction level: value-types `DocumentCode`, `Reference` and entity-type `Customer`.

With value-types there is no risk of confusing `Reference` with `DocumentCode`, or `Amount` with `Discount Percentage`.

Base Classes

Value Types extend one of the following domain classes: `DomainBoolean`, `DomainDate`, `DomainDateTime`, `DomainDecimal`, `DomainInteger`, `DomainLong`, `DomainString`, `DomainText`, `DomainTime` or implement the `DomainEnumerate` interface .

With the exception of `DomainEnumerate`, these domain types wrap `Java` primitives

Domain Type	**Wrapped `Java` Type**	**Meta Type**
`DomainBoolean`	`Boolean`	`BooleanMeta`
`DomainDate`	`LocalDate`	`DateMeta`

Domain Type	Wrapped Java Type	Meta Type
DomainDateTime	LocalDateTime	DateTimeMeta
DomainDecimal	BigDecimal	DecimalMeta
DomainEnumerate	Enumerate	EnumerateMeta
DomainInteger	BigInteger	IntegerMeta
DomainLong	Long	LongMeta
DomainShort	Short	ShortMeta
DomainString	String	StringMeta
DomainText	String	TextMeta
DomainTime	LocalTime	TimeMeta

The *Meta Type* provides additional information, restricting the domain values:

Meta Type	Attributes
LocalDateTime	TimePrecision (min., sec., mil.)
DecimalMeta	precision, scale, rounding
IntegerMeta	min./max. value
LongMeta	min./max. value
ShortMeta	min./max. value
StringMeta	length, uppercase
TimeMeta	TimePrecision (min., sec., mil.)

Domain Enumerate is an Interface as Enumerates in Java do not support inheritance.

Value type objects are immutable, they can benefit from caching ('flyweight' design-pattern) using static constructors.

Domain Boolean

Boolean Value Types extend DomainBoolean and must implement method createInstance().

```
class Confirmed extends DomainBoolean<Confirmed>{

    public Confirmed(final Boolean value) {
        super(value);
    }

    @Override
    protected Confirmed createInstance(final Boolean value) {
        return new Confirmed(value);
    }
}
```

Value

```
final Confirmed confirmed = new Confirmed(true);

assertEquals(true, confirmed.getValue());
```

Usage

```
final Confirmed confirmed = new Confirmed(true);

assertEquals(true, confirmed.getValue());
assertFalse(confirmed.isFalse());
assertTrue(confirmed.isTrue());
assertTrue(confirmed.getValue());
```

Equality

```
final Confirmed confirmed1 = new Confirmed(true);
final Confirmed confirmed2 = new Confirmed(true);

assertEquals(confirmed1, confirmed2);
```

```
final Confirmed confirmed = new Confirmed(true);

final Confirmed unconfirmed1 = confirmed.negate();
final Confirmed unconfirmed2 = confirmed.negate();

assertEquals(unconfirmed1, unconfirmed2);
```

Negation

```
final Confirmed confirmed = new Confirmed(true);
final Confirmed unconfirmed = confirmed.negate();

assertTrue(unconfirmed.isFalse());
assertFalse(unconfirmed.isTrue());
assertFalse(unconfirmed.getValue());
assertNotEquals(confirmed, unconfirmed);
```

Domain Date

Date Value Types extend `DomainDate` and must implement method `createInstance()`.

For example:

```
class InvoiceDate extends DomainDate<InvoiceDate>{

    public InvoiceDate(final LocalDate value) {
        super(value);
    }

    public InvoiceDate(final int year, final int month, final int dayOfMonth) {
        super(year, month, dayOfMonth);
    }

    @Override
    protected InvoiceDate createInstance(final LocalDate value) {
        return new InvoiceDate(value);
    }
}
```

Value

```
InvoiceDate invoiceDate = new InvoiceDate(2021, 12, 12);

assertEquals(LocalDate.of(2021, 12, 12), invoiceDate.getValue());
```

Usage

```
InvoiceDate d1 = new InvoiceDate(2021, 12, 12);
InvoiceDate d2 = d1.plusDays(5);
InvoiceDate d3 = d1.minusDays(5);

assertEquals(LocalDate.of(2021, 12, 17), d2.getValue());
assertEquals(LocalDate.of(2021, 12, 7), d3.getValue());

assertFalse(d2.isBefore(d1));
assertTrue(d2.isAfter(d1));

assertTrue(d3.isBefore(d1));
assertFalse(d3.isAfter(d1));
```

Compare

```
final InvoiceDate d1 = new InvoiceDate(LocalDate.of(2021, 12, 12));
final InvoiceDate d2 = d1.plusDays(5);
final InvoiceDate d3 = d1.minusDays(5);

assertTrue(d2.compareTo(d1)>0);
assertTrue(d3.compareTo(d1)<0);

final InvoiceDate d4 = new InvoiceDate(LocalDate.of(2021, 12, 12));
assertTrue(d1.compareTo(d4)==0);
```

Equality

```
final InvoiceDate d1 = new InvoiceDate(LocalDate.of(2021, 12, 12));
final InvoiceDate d2 = new InvoiceDate(LocalDate.of(2021, 12, 12));
assertEquals(d1, d2);
```

Domain DateTime

DateTime Value Types extend `DomainDateTime` and must implement method `createInstance()`.

The constructor requires a meta-object `DateTimeMeta` to specify the precision: minutes, seconds or milliseconds.

```
class TimeStamp extends DomainDateTime<TimeStamp>{

    public TimeStamp(final LocalDateTime value) {

        super(DateTimeMeta.create(TimePrecision.MINUTES), value);
    }

    @Override
    protected TimeStamp createInstance(final LocalDateTime value) {
        return new TimeStamp(value);
    }
}
```

Value

```
TimeStamp timestamp = new TimeStamp(2021,8,1,22,00,30);

assertEquals(LocalDateTime.of(2021,8,1,22,00,30), timestamp.getValue());
```

Usage

```
TimeStamp d1 = new TimeStamp(2021,8,1,22,00,00);
TimeStamp d2 = d1.plusMinutes(5);
TimeStamp d3 = d1.minusHours(5);
TimeStamp d4 = d1.plusSeconds(10);
TimeStamp d5 = d1.minusSeconds(20);

assertEquals(new TimeStamp(2021,8,1,22,05,00), d1);
assertEquals(new TimeStamp(2021,8,1,22,05,00), d2);
assertEquals(new TimeStamp(2021,8,1,17,00,00), d3);
assertEquals(new TimeStamp(2021,8,1,22,00,10), d4);
assertEquals(new TimeStamp(2021,8,1,21,59,40), d5);
```

Compare

```
assertTrue(d2.isAfter(d1));
assertTrue(d3.isBefore(d1));
assertTrue(d4.isAfter(d1));
assertTrue(d5.isBefore(d1));
```

Domain Decimal

Decimal Value Types extend `DomainDecimal` and must implement method `createInstance()`.

The constructor requires a meta-object `DecimalMeta` to specify precision, scale and rounding.

```
public class Amount extends DomainDecimal<Amount> {

    private static final DecimalMeta META = DecimalMeta.create(7, 2, RoundingMeta
.HALF_UP);

    public Amount(final BigDecimal value) {
        super(META, value);
    }

    @Override
    public Amount createInstance(final BigDecimal value) {
        return new Amount(value);
    }
}
```

Override `DomainDecimal` constructors for ease of use:

```
public Amount(final double value) {
    super(META, value);
}

public Amount(final String value) {
    super(META, value);
}

public Amount(final DomainDecimal<?> domainValue) {
    super(META, domainValue.getValue());
}
```

Value

```
Amount quantity = new Amount(3.50);

assertEquals(new BigDecimal("3.50"), quantity.getValue());
```

Scale

Domain decimals are rounded upon creation to ensure that no invalid objects can exist.

```
// Increase precision
assertEquals(new Amount("0.00"), new Amount("0.0"));
assertEquals(new Amount("1.00"), new Amount("1"));
assertEquals(new Amount("1.20"), new Amount("1.2"));

// Round down
assertEquals(new Amount("1.23"), new Amount("1.233"));

// Round up
assertEquals(new Amount("1.24"), new Amount("1.237"));
```

Precision

```
// Precision exceeds 7 digits
 assertThrows(DomainValueException.class, ()->{
    new Amount("1234567.89");
});
```

Usage

We can now add additional constructors for ease of use and methods for business logic.

First a new constructor for `Amount` to create new instances based on any domain decimal:

```
public Amount(final DomainDecimal<?> value) {
    this(value.getValue());
}
```

and two new Decimal types, `Quantity`:

```
    class Quantity extends DomainDecimal<Quantity> {

        public Quantity(final BigDecimal value) {
            super(DecimalMeta.create(5, 2, MRounding.HALF_UP), value);
        }

        @Override
        protected Quantity createInstance(final BigDecimal value) {
            return new Quantity(value);
        }
    }
```

and `Price`:

```
class Price extends DomainDecimal<Price> {

    public Price(final BigDecimal value) {
        super(DecimalMeta.create(5, 3, MRounding.HALF_UP), value);
    }

    @Override
    protected Price createInstance(final BigDecimal value) {
        return new Price(value);
    }

    public Amount multiply(final Quantity quantity) {
        return new Amount(super.multiply(quantity));
    }
}
```

Value Types are plain `Java` classes and can be extended with extra functionality, like the `multiply` method in the `Price` class. This method takes a quantity, multiplies it with the price and returns an amount using the new amount-constructor.

Given these new value types, calculations are possible using only the business concepts Amount, Price and Quantity:

```
final Price price = new Price("13.3329");
final Quantity quantity = new Quantity("3.50");

final Amount amount = price.multiply(quantity);

assertEquals(price, new Price(13.333));
assertEquals(amount, new Amount(46.67));
```

It is important to notice that two roundings were applied. The price is rounded to `13.333` before calculation. And after multiplication the result is rounded from `46.6655` to `46.67`.

Automatic rounding must be used with care as a loss of precision can result in imprecise results.

Validation

Value Types can be validated by adding checks in the constructor. For example, disallowing negative prices:

```
public Price(final BigDecimal value) {
    super(DecimalMeta.create(7, 2, MRounding.HALF_UP), value);

    if(isNegative()) {
        throw DomainValueException.create("Negative price not allowed: "+this);
    }
}
```

This will cause an exception in all business logic whenever a negative price is created.

```
assertThrows(DomainValueException.class, ()->{
    final Price price = new Price(new BigDecimal("-3.59"));
});
```

Convenience methods

Besides the standard methods `add`, `substract`, `multiply`, `divide`, `max` and `min`, domain decimals include additional convenience methods:

```
final Amount amount = new Amount(-1.23);

assertEquals(new Amount(1.23), amount.absolute());
assertEquals(new Amount(1.23), amount.negate());
assertEquals(true, amount.isNegative());
assertEquals(false, amount.isPositive());
```

```
final Amount amount = new Amount(1);

assertEquals(false, amount.isZero());
assertEquals(true, amount.isNonZero());
assertEquals(true, amount.isOne());
assertEquals(false, amount.isNegativeOne());
```

Equality

Domain Decimal equality is based on the underlying BigDecimal value.

```
Amount a1 = new Amount(1.23);
final Amount a2 = new Amount(1.2349);①

assertEquals(a1, a2);
```

① amount is rounded **down** on instantiation

```
Amount a1 = new Amount(1.24);
final Amount a2 = new Amount(1.2350);①

assertEquals(a1, a2);
```

① amount is rounded **up** on instantiation

Values types with the same precision and scale can be equal.

```
Amount a1 = new Amount(1.23);
Price p1 = new Price(1.23);

assertNotEquals(a1, p1);
assertEquals(a1, new Amount(p1));
```

Extend the `DomainDecimal` value type with domain specific constructors:

```
public static Amount of(
        final Quantity quantity,
        final Price price) {

    return new Amount(
        quantity.getValue()
        .multiply(price.getValue()));
}
```

Domain Integer / Long / Short

Integer Value Types extend `DomainInteger` and must implement method; `createInstance()`.

The constructor requires a meta-object `IntegerMeta` to specify lower- and upper-limit.

```
class PortNumber extends DomainInteger<PortNumber>{

    public PortNumber(final Integer value) {
        super(IntegerMeta.create(0, 65535), value);
    }

    @Override
    protected PortNumber createInstance(final Integer value) {
        return new PortNumber(value);
    }
}
```

Value

```
PortNumber portNumber = new PortNumber(80);

assertEquals(80, portNumber.getValue());
```

Usage

```
PortNumber d1 = new PortNumber(80);
PortNumber d2 = new PortNumber(8000);
PortNumber d3 = d1.add(d2);

assertEquals(new PortNumber(8080), d3);
assertFalse(d1.isNegative());
assertFalse(d1.isZero());
```

Note that an exception is thrown if the value is not within the declared range:

```
assertThrows(DomainValueException.class, ()->{
    new PortNumber(-1);
});
```

Calculations

```
PortNumber n1 = new PortNumber(1);
PortNumber n2 = new PortNumber(15);
PortNumber n3 = new PortNumber(4);

assertEquals(new PortNumber(3), n2.divide(n3));
assertEquals(new PortNumber(60), n2.multiply(n3));
assertTrue(n1.isOne());
```

Notice that the result of `divide()` is truncated to an integer value.

Value types `DomainShort` and `DomainLong` have similar semantics as `DomainInteger`.

Domain String / Text

String Value Types extend `DomainString` and must implement method `createInstance()`.

The constructor requires a meta-object `StringMeta` to specify length and case.

```
class Description extends DomainString<Description>{

    public Description(final String value) {
        super(StringMeta.create(60, false), value);
    }

    @Override
    protected Description createInstance(final String value) {
        return new Description(value);
    }
}
```

Value

```
Description description = new Description("abc");

assertEquals("abc", description.getValue());
```

Usage

```
final Description d = new Description("abcdefg");

assertEquals(new Description("abcdefg_hij"),d.append("_hij"));
assertEquals("abc", d.getLeft(3));
assertFalse(d.isEmpty());
assertTrue(d.isNotEmpty());
```

Trailing spaces are removed.

```
Description d = new Description("abc   ");
assertEquals(3, d.length());
assertEquals("abc", d.getValue());
```

Leading-spaces are not removed by default. Some (legacy) systems use leading-spaces to right-align field values.

Strings are not truncated, if a value exceeds the domain, an exception will be thrown:

```
assertThrows(DomainValueException.class, ()->{
    new Description("abcdefg".repeat(30));
});
```

Equality

Equality is based on the underlying String value.

```
DocumentCode documentCode = new DocumentCode("abc");
Description description = new Description("abc");

assertTrue(documentCode.equals(description));
```

Value type `DomainText` is not described here as it works the same as `DomainString`, except for the limitless text length.

Domain Time

Time Value Types extend `DomainTime` and must implement method `createInstance()`.

The constructor requires a meta-object `TimeMeta` to specify the precision: minutes, seconds or milliseconds.

For example, the declaration of a new `StartTime` type:

```
class StartTime extends DomainTime<StartTime>{

    public StartTime(final LocalTime value) {

        super(TimeMeta.create(TimePrecision.MINUTES), value);
    }

    @Override
    protected StartTime createInstance(final LocalTime value) {
        return new StartTime(value);
    }
}
```

override a constructor for convenience:

```
public StartTime(final int hour, final int minute) {
    super(hour, minute);
}
```

Value

```
StartTime startTime = new StartTime(20,00);

assertEquals(LocalTime.of(20,00), startTime.getValue());
```

Usage

```
StartTime d1 = new StartTime(20,00);
StartTime d2 = d1.plusMinutes(5);
StartTime d3 = d1.minusHours(5);

assertEquals(new StartTime(20,05), d2);
assertEquals(new StartTime(15,00), d3);
```

Compare

```
assertTrue(d2.isAfter(d1));
assertTrue(d3.isBefore(d1));
```

Domain Entities

Overview

Domain Entities extend class `DomainEntity` and must implement method `getBusinessKey()` and `getEntityKey()`.

```
public class Document implements DomainEntity{

    private final EntityKey entityKey;

    private DocumentCode code;
    private Description description;

    public Document(
            final EntityKey entityKey,
            final DocumentCode code,
            final Description description) {
        super();
        this.entityKey = entityKey;
        this.code = code;
        this.description = description;
    }

    @Override
    public EntityKey getEntityKey() {
        return entityKey;
    }

    @Override
    public BusinessKey getBusinessKey() {
        return BusinessKey.of(code.getValue());
    }
    //...
```

EntityKey

The `EntityKey` is an internal technical identifier of the entity, normally a database record `id` or `UUID` (Universal Unique Identifier).

BusinessKey

The `BusinessKey` is an external domain identifier of the entity which is communicated to users of the application.

Getters/Setters

Getters and setters are defined in terms of Domain Values.

```
public DocumentCode getCode() {
    return code;
}

public void setCode(final DocumentCode code) {
    this.code = code;
}
```