



# **Clinical Quality Language (CQL) Basics**

---

**Shanna Hartman**  
Centers for Medicare & Medicaid Services

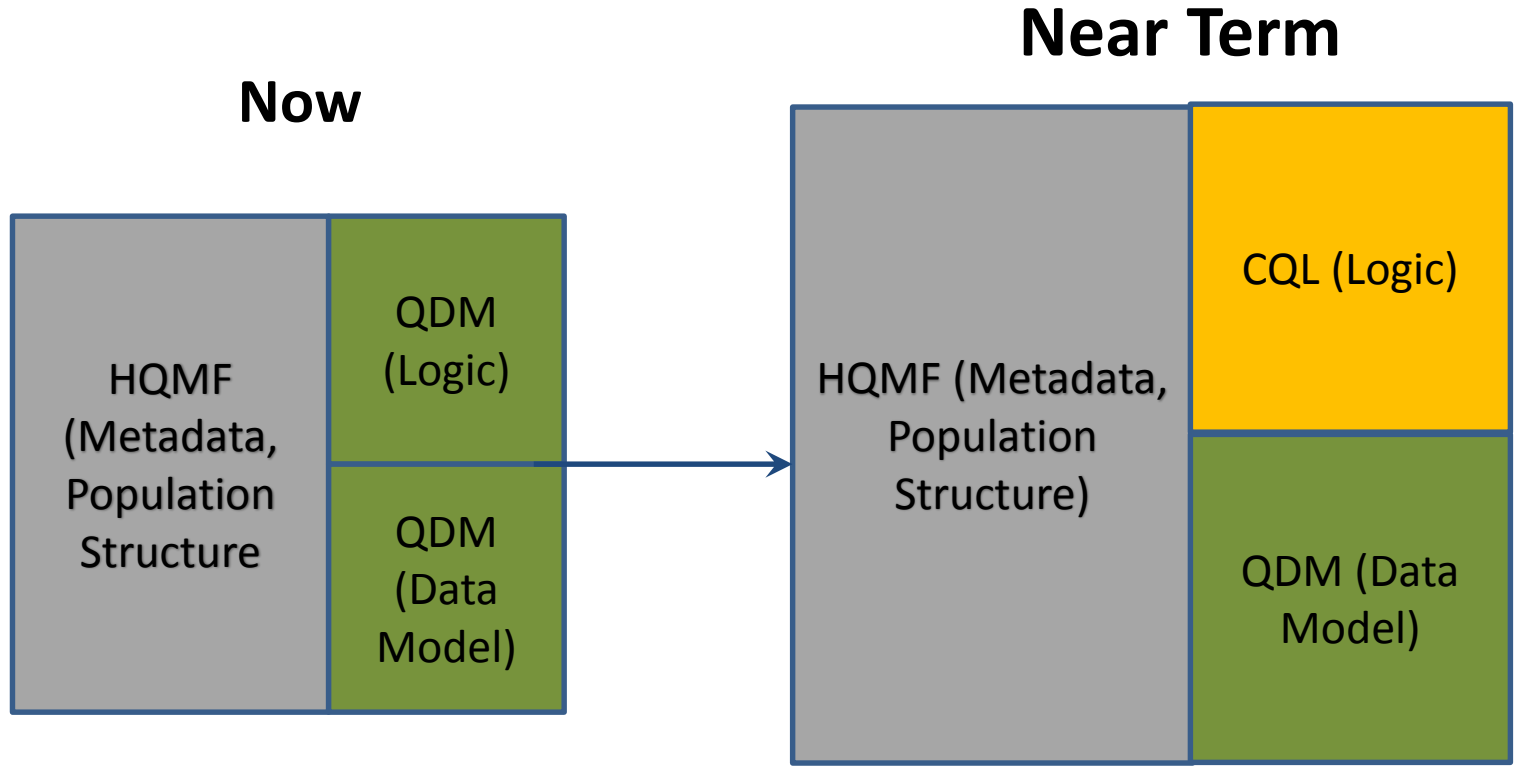
**Deborah Krauss**  
Centers for Medicare & Medicaid Services (CMS)

**Bryn Rhodes**  
ESAC, Inc.

# CQL Basics: Agenda

- Welcome and Background
- CQL Language Tour
  - Accessing Clinical Data
  - Using Queries
  - Computation
  - Date/Times, Intervals, and Timing Phrases
  - Combining Queries
  - Aggregate Computation

# CQL Basics: Evolving eCQM Standards



**Definitions:**

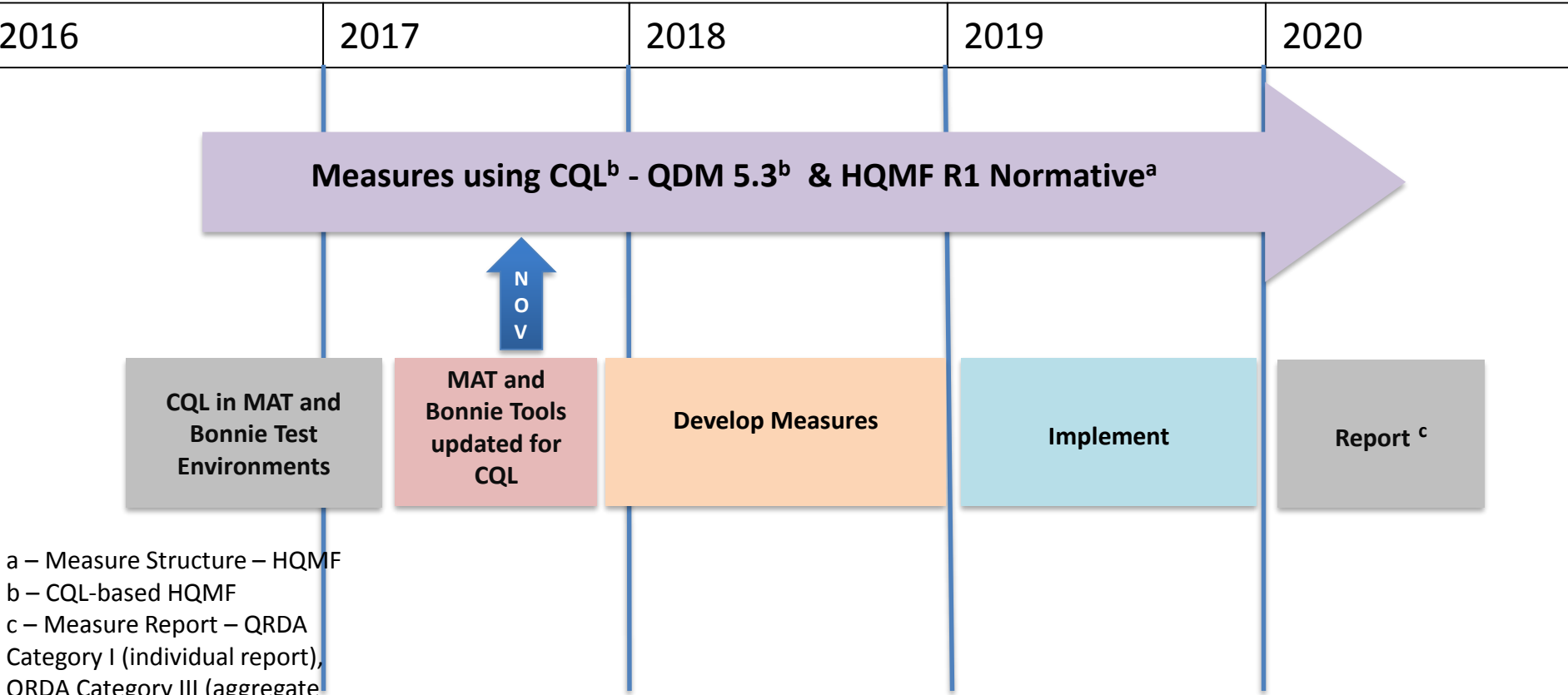
**HQMF** – Health Quality Measure Format

**CQL** – Clinical Quality Language

**QDM** – Quality Data Model

# CQL Basics: Proposed Timeline For Updating Standards

## Measure Development – Expected Timelines



a – Measure Structure – HQMF  
 b – CQL-based HQMF  
 c – Measure Report – QRDA Category I (individual report), QRDA Category III (aggregate report)  
 Updated 12/8/2017

---

# CQL LANGUAGE TOUR

# CQL Basics: Clinical Quality Language (CQL)

- Health Level 7(HL7) standard designed to:
  - Enable automated point-to-point sharing of executable clinical knowledge
  - Provide a clinically focused, author-friendly, and human-readable language
- Currently a Draft Standard for Trial Use (DSTU) publication
  - [http://www.hl7.org/implement/standards/product\\_brief.cfm?product\\_id=400](http://www.hl7.org/implement/standards/product_brief.cfm?product_id=400)

# CQL Basics: Accessing Clinical Data

- Clinical data models contain “statements” of clinical data, e.g.,
  - Patient had a routine check-up on April 3<sup>rd</sup>
  - Patient was administered an antibiotic
  - Patient had an appendectomy
  - Patient was diagnosed with Type II Diabetes

# CQL Basics: Accessing Clinical Data (cont'd)

- Statements can be organized into different *types, e.g.,*

Patient had a routine check-up on April 3<sup>rd</sup>  
Patient was administered an antibiotic  
Patient had an appendectomy  
Patient was diagnosed with Type II  
Diabetes

Encounter  
Medication, Administered  
Procedure, Performed  
Diagnosis



## CQL Basics: Accessing Clinical Data (cont'd)

- Within these types, different kinds of statements can be represented with *codes* from *code systems*.
- Within Encounter, e.g.,

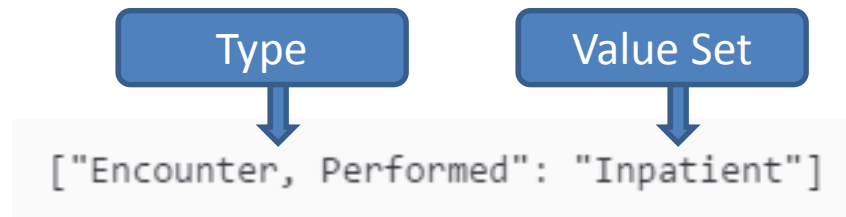
Patient had a routine check-up on April 3 <sup>rd</sup>	SNOMEDCT 185349003
Patient was admitted to the ED	SNOMEDCT 4525004
Patient was admitted for elective surgery	SNOMEDCT 8715000

## CQL Basics: Accessing Clinical Data (cont'd)

- The codes describing different kinds of statements are then grouped with *value sets*, allowing classes of specific kinds of statements to be referenced, e.g.,

Patient had a routine check-up on April 3 <sup>rd</sup>	Encounter Inpatient
Patient was admitted to the ED	Emergency Department Visit
Patient was admitted for elective surgery	Elective Encounter

# CQL Basics: CQL Retrieve



This *retrieve* expression results in only the highlighted encounters, because they have codes that match the “Inpatient” value set.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
4	SNOMEDCT 305338009	[2015-05-13T10:00, 2015-05-15T18:00]

# CQL Basics: Filtering with Where

```
["Encounter, Performed": "Inpatient"] Encounter
  where Encounter.relevantPeriod during "Measurement Period"
```

Introducing an *alias*, **Encounter** in this case, allows you to reference elements of the statement for further filtering using a *where clause*. The above filter results in only the highlighted rows, because they are *during* the “Measurement Period” (2015 year).

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]

# CQL Basics: Relationships (with)

```
["Encounter, Performed": "Inpatient"] Encounter
  with ["Laboratory Test, Performed": "Streptococcus Test"] LabTest
  such that LabTest.resultDateTime during Encounter.relevantPeriod
```

The *with clause* allows you to define relationships with other data based on specific criteria.

In this case, only the October encounter is returned, because it has a LabTest that resulted during the encounter.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]

id	code	result	resultDateTime	status
1	LOINC 68954-7	positive	2015-10-14T14:00	completed
2	LOINC 6559-9	negative	2015-10-12T17:00	completed

# CQL Basics: Relationships (without)

```
["Encounter, Performed": "Inpatient"] Encounter
  without ["Laboratory Test, Performed": "Streptococcus Test"] LabTest
  such that LabTest.resultDateTime during Encounter.relevantPeriod
```

Statements can also be excluded based on relationships using the *without clause*.

In this case, the October encounter is excluded, because it has a LabTest that resulted during the encounter.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]

id	code	result	resultDateTime	status
1	LOINC 68954-7	positive	2015-10-14T14:00	completed
2	LOINC 6559-9	negative	2015-10-12T17:00	completed

# CQL Basics: Shaping Results with Return

```
["Encounter, Performed": "Inpatient"] Encounter
  return { relevantPeriod: Encounter.relevantPeriod }
```

You can return only a subset of the elements in a statement using the *return clause*.

In this case, only the relevantPeriod element is returned.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]

# CQL Basics: Ordering Results with Sort

```
[ "Encounter, Performed": "Inpatient" ] Encounter  
  sort by start of relevantPeriod
```

You can order the results using the *sort clause*.

In this case, the result is sorted by the start of the `relevantPeriod` element, ascending.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]



# CQL Basics: Naming Expressions

```
define "Sorted Encounters":
  ["Encounter, Performed": "Inpatient"] Encounter
  sort by start of relevantPeriod
```

You can name any expression so that it can be reused in subsequent expressions using the *define declaration*.

In this case, the result of “Sorted Encounters” is now the same as the result of the defined expression.

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]

# CQL Basics: Picking Items from Results

```
First("Sorted Encounters")
```

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]

```
Last("Sorted Encounters")
```

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]

The results of retrieves and queries are lists, so you can pick items based on order using *First()* and *Last()*.

Because “Sorted Encounters” is ordered by the start of the relevantPeriod, *First()* returns the oldest encounter, while *Last()* returns the most recent.

## CQL Basics: Picking Items (cont'd)

```
"Sorted Encounters"[0]
```

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]

```
"Sorted Encounters"[1]
```

id	code	relevantPeriod
1	CPT 99392	[2014-12-13T13:00, 2014-12-13T13:00]
3	CPT 99392	[2015-03-13T08:00, 2015-03-13T08:15]
2	CPT 99391	[2015-10-14T14:00, 2015-10-14T14:00]

You can also use the *indexer* ([ ]) to pick out any item by its index in the list.

Indexes in CQL are 0-based, so the first item is index 0, the second item is index 1, and so on.

Note that whenever you're performing operations that rely on the order of elements in the list, be sure to use a *sort clause* to get the appropriate ordering.

# CQL Basics: Strings

```
'John Doe'  
'John O\'Mally'  
'John Doe' = 'john doe' // false  
'Deer' < 'Doe'           // true
```

CQL supports strings using single-quotes (').

You can *escape* characters (such as single-quotes, tabs, carriage returns, and line feeds, using standard escape characters.

CQL supports string comparison for all the comparison operators (=, !=, <=, >=, <, and >).

String comparison is case-sensitive, and based on the Unicode value of each character.

# CQL Basics: Numbers and Calculation

```
5
5.0
5 + 5.0    // 10.0
5.0 = 5.00 // true
```

```
2 + 5 * 10    // 52
(2 + 5) * 10  // 70
10 / 2        // 5.0
10 div 2      // 5
10 mod 2      // 0
```

CQL supports *Integers* (whole numbers), and *Decimals*.

In calculations and comparisons, integers are implicitly converted to decimals when necessary.

Comparison of decimals ignores precision.

CQL uses standard mathematical operator precedence.

Use parentheses to force precedence.

Division in CQL always returns a decimal, use *div* to perform integer division.

The *mod* operator returns the remainder of an integer division.

# CQL Basics: Rounding and Exponents

```
Round(5.5) // 6
Round(5.55, 1) // 5.6
Truncate(5.5) // 5
Truncate(-5.5) // -5
Floor(5.5) // 5
Floor(-5.5) // -6
Ceiling(5.5) // 6
Ceiling(-5.5) // -5
```

CQL supports standard rounding, 0.5 and above rounds up, 0.4 and below rounds down. The second argument, if supplied, specifies the precision of the result.

*Truncate()* returns the integer component of a decimal.

*Floor()* returns the greatest integer less than a decimal.

*Ceiling()* returns the least integer greater than a decimal.

```
5 ^ 2 // 25
25 ^ 0.5 // 5
Log(25, 5) // 2
Log(5, 25) // 0.5
Ln(10) // 2.30258209288405
Exp(2.30258509288405) // 10
```

CQL supports exponents and roots with  $^$ .

Logarithms to a given base use *Log()*.

Natural logarithms use *Ln()* and *Exp()*.

# CQL Basics: Quantities

Quantities in CQL are a number followed by a UCUM unit.

CQL supports arithmetic and comparison operators for quantities.

Implementations are required to respect units, but not necessarily conversions between units.

Arithmetic operators, in particular, must return quantities with appropriate units, but not necessarily converted.

An implementation may throw a run-time error for an unsupported unit conversion operation.

```
25 'mg'  
100 'cm2'  
1 'm' = 100 'cm' // true  
10 'cm' * 10 'cm' // 100 'cm2'
```

# CQL Basics: DateTime and Time

```
@2014-01-25
@2014-01-25T14:30:14.5

@T12:00:00.0Z
@T14:30:14.5-07:00

@2014
@2014-01
@T14
@T14:30
```

CQL supports `dateTime`, a point-in-time on the Western calendar, specified with integers for year, month, day, hour, minute, second, and millisecond, plus a timezone.

CQL also supports `Time`, a point-in-time in a 24-hour period, specified with integers for hour, minute, second, and millisecond, plus timezone.

Both `dateTime` and `Time` support partial values, but only for trailing precisions (i.e., if you specify a day, you must also specify a year and month).

If not supplied, timezone is assumed based on the evaluation context.



# CQL Basics: DateTime and Time (cont'd)

```
DateTime(2014, 7, 5)  
Time(14, 30)
```

```
date from @2014-01-25T14:30:14 // 2014-01-25  
time from @2014-01-25T14:30:14 // T14:30:14  
year from @2014-01-25 // 2014
```

```
Now()  
Today()  
TimeOfDay()
```

CQL also supports construction of `dateTime` and `Time` values as expressions.

You can use *date from* to extract the date (with no time components) from a `dateTime` value.

You can use *time from* to extract the time from a `dateTime` value.

You can use the name of a component to extract it from a `dateTime` or `Time` value.

*Now()*, *Today()*, and *TimeOfDay()* return the `dateTime`, `Date`, and `Time`, respectively, of the evaluation context.

# CQL Basics: Date Comparison

```
@2014-01-15 = @2014-02-15 // false
@2014-01-15 < @2014-02-15 // true
@2014-01-15 <= @2014-02-15 // true
@2014-01-15 same year as @2014-02-15 // true
@2012-01-15 same year or before @2014-02-15 // true
@2012-01-15 before year of @2014-02-15 // true
```

You can compare dateTime and Time values using the standard comparison operators: =, !=, <=, >=, <, and >.

You can also perform precision-based comparisons using *same as*, *before/after of*, and *same or before/after*.

# CQL Basics: Date Arithmetic

```
1 day
2 years
30 minutes
1 'd'
2 'a'
30 'min'
```

CQL supports time-valued quantities with the name (singular or plural) of the precision as the unit.

UCUM units can also be used (with quotes).

```
Today() - 1 year
@2014-02-01T14:30 + 30 minutes // 2014-02-01T15:00
@2014 + 24 months // 2016
```

Durations can then be added to or subtracted from dateTime and Time values, with the expected semantics for durations with variable days such as years and months.

# CQL Basics: Computing Duration and Difference

```
duration in months between @2014-01-31 and @2014-02-01 // 0
```

The *duration in..between* operator determines the number of whole periods between two dateTime or Time values.

This expression returns 0 because there are no whole months between the two dates.

```
difference in months between @2014-01-31 and @2014-02-01 // 1
```

The *difference in..between* operator determines the number of boundaries crossed between two dateTime and Time values.

This expression returns 1 because 1 month boundary was crossed between the two dates.

# CQL Basics: Intervals

```
Interval[3, 5)      // 3, 4
Interval[3.0, 5.0) // >= 3.0, < 5.0
Interval[@2014-01-01T00:00:00.0, @2015-01-01T00:00:00.0]
```

CQL supports Intervals for numbers and date/time values.

Intervals use standard mathematical notation to indicate open and closed (i.e., whether the endpoint is included in (closed) or excluded from (open) the interval).

```
Interval[3, 5) contains 4 // true
4 in Interval[3, 5)      // true
```












You can test for membership with *contains* and *in*, and you can determine the boundaries of an interval using *start of* and *end of*.

```
start of Interval[3, 5) // 3
end of Interval[3, 5)  // 4
```

You can determine the width of an interval using *width of*.

```
width of Interval[3, 5) // 2
```

# CQL Basics: Comparing Intervals

before	
meets before	
overlaps before	
includes	
starts	
same as (=)	
ends	
included in (during)	
overlaps after	
meets after	
after	

# CQL Basics: Timing Phrases

CQL also supports timing phrases that make it easier to express precise relationships between intervals using natural language.

The *before* and *after* operators can have a prefix of *starts* or *ends*, and a suffix of *start* or *end*. For example,

```
IntervalX starts before start IntervalY
```

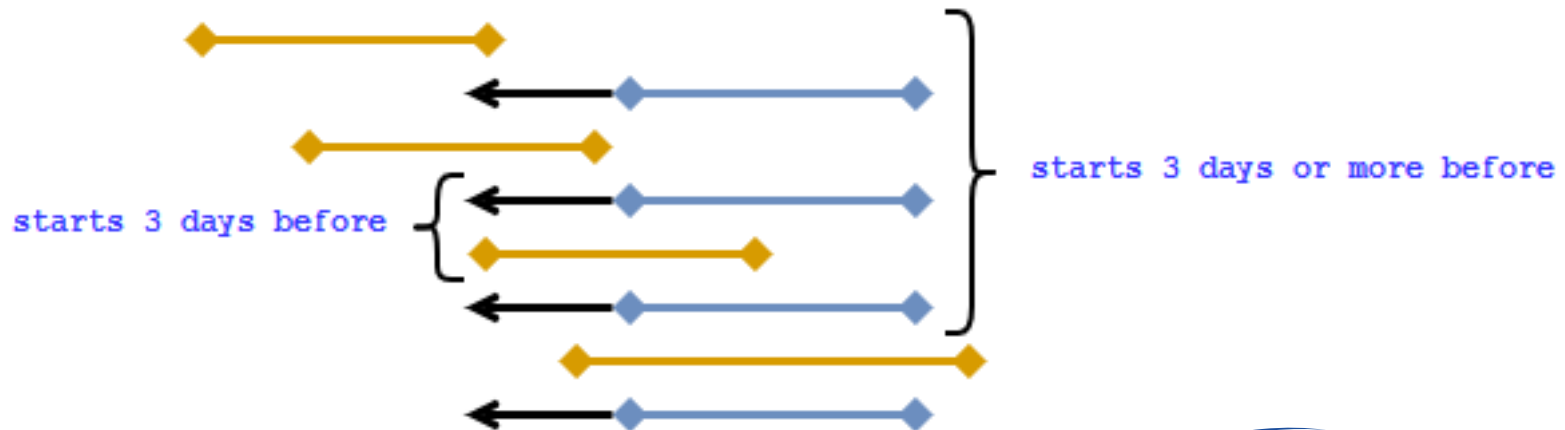


## CQL Basics: Timing Phrases (cont'd)

The *before* and *after* operators can also take an offset that indicates how far away a given relationship should be.

This offset can be absolute, indicating that the boundary of the interval must be on the offset, or it can be relative, indicating that the boundary must be at least on the offset.

```
IntervalX starts 3 days before start IntervalY
IntervalX starts 3 days or more before start IntervalY
```

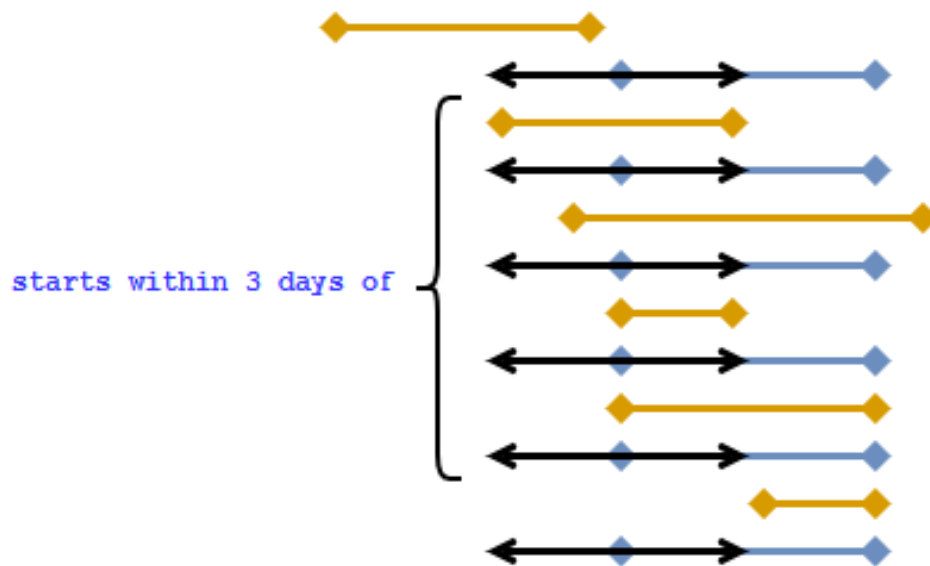




## CQL Basics: Timing Phrases (cont'd)

You can also specify a range for the boundary relationship using the *within..of* operator.

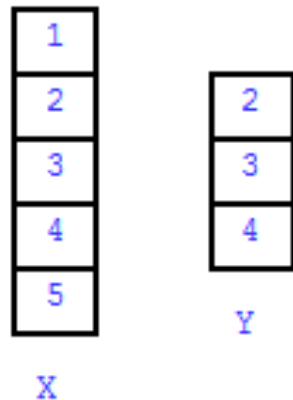
```
IntervalX starts within 3 days of start IntervalY
```



# CQL Basics: List Operations

You can test for membership of items in a list using the *contains* and *in* operators.

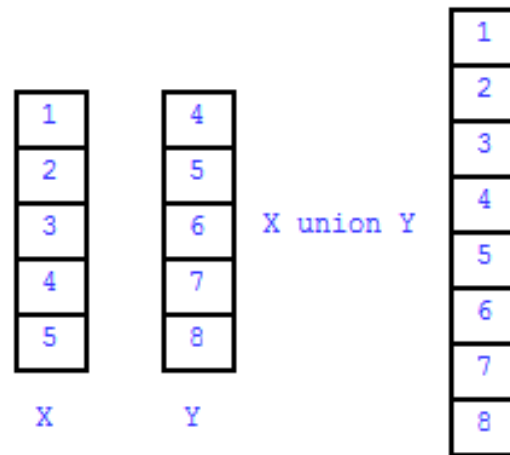
You can compare lists using equality (=), and the *includes* and *included in* operators.



```
X contains 3 // true
3 in X // true
X includes Y // true
Y included in X // true
```

# CQL Basics: Union

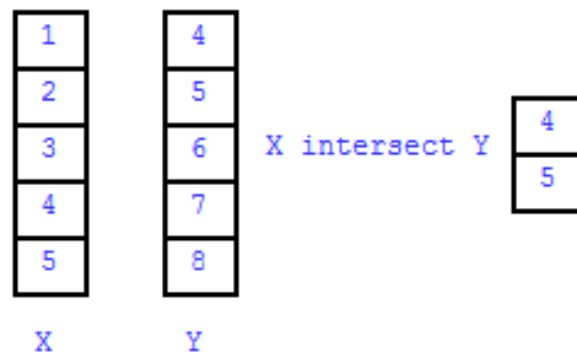
The *union* operator combines two lists, eliminating duplicates.



NOTE: In the current specification (CQL 1.1), union does not eliminate duplicates, so a *distinct* must be used. However, this is a DSTU comment to change this behavior to support the more intuitive duplicate elimination semantics.

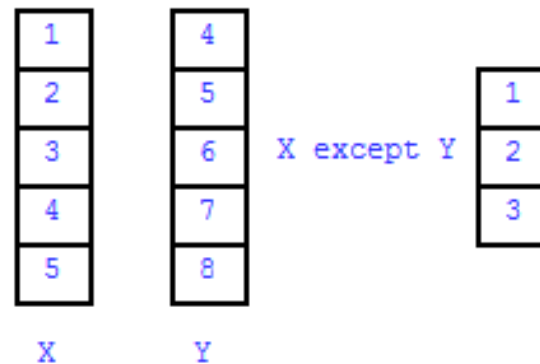
# CQL Basics: Intersect

The *intersect* operator results in a list containing only the elements that appear in both lists.



# CQL Basics: Except

The *except* operator results in a list containing only the elements of the first list that are not present in the second list.



# CQL Basics: Resources

- CQL Specification - CQL Release 1, Standard for Trial Use (STU) 2
  - [http://www.hl7.org/implement/standards/product\\_brief.cfm?product\\_id=400](http://www.hl7.org/implement/standards/product_brief.cfm?product_id=400)
- eCQI Resource Center
  - CQL Space, including the QDM v5.3 and v5.3 Annotated
    - <https://ecqi.healthit.gov/cql>
  - Check the eCQI Resource Center Events page and CQL Educational Resources page for more information
    - <https://ecqi.healthit.gov/ecqi/ecqi-events>
    - <https://ecqi.healthit.gov/cql/cql-educational-resources>



# CQL Basics: Resources (cont'd)

- CQL Formatting and Usage Wiki
  - <https://github.com/esacinc/CQL-Formatting-and-Usage-Wiki/wiki>
- CQL GitHub Tools Repository
  - [https://github.com/cqframework/clinical\\_quality\\_language](https://github.com/cqframework/clinical_quality_language)
- Measure Authoring Tool
  - <https://www.emeasuretool.cms.gov/>
- Bonnie Testing Tool
  - <https://bonnie.healthit.gov/>
- To submit an issues ticket for CQL, please visit the ONC JIRA site
  - <https://oncprojecttracking.healthit.gov/support/projects/CQLIT>