QueryTests

TCK20: JDO2 Query Test Cases

JDOQL 2.0

JDO 2.0 adds the following methods to the Query API:

- setResult
- setGrouping
- setUnique
- setResultClass
- setRange
- setUnique
- setUnmodifiable
- isUnmodifiable
- setExtensions
- addExtension

JDO 2.0 extensions of the JDO query language JDOQL:

- Single string JDOQL
- Result specification
  - Projections of fields and relationships
  - One or more result expressions
  - Distinct results
  - Unique query result
  - Default result class for one or more result expressions
  - User defined result class
  - Naming of result expressions
- Aggregate functions MIN, MAX, SUM, AVG, and COUNT
- Grouping of query result
  - One or more grouing expressions
  - Having clause
- New methods in Query filters:
  - Map support: get(Object), containsKey(Object), containsValue(Object), isEmpty()
  - Additional string methods: toLowerCase(), toUpperCase(), indexOf(String), indexOf(String, int), matches(String), substring(int), substring (int, int)
  - Support for other methods: Math.abs(numeric), Math.sqrt(numeric), JDOHelper.getObjectId(Object)
- New operators %(modulo) and instanceof
- Support for implicit parameters
- Support for implicit variables
- Deletion by query

NewQueryTests New TCK Query Tests

Package names of all query test classes start with `org.apache.jdo.query.`.

Package names of all pc classes start with `org.apache.jdo.tck.pc.`.

Package names of all result classes start with `org.apache.jdo.tck.query.result.classes.`.

Language Extensions

Keywords

Assertion	JDOQL	Testclass	Comment
A14.4-6: Keywords must not be used as package names, class names, parameter names, or variable names in queries.	`SELECT INTO range.PersonResult company.Person`	`jdoql.keywords.InvalidUseOfKeywords`	Negative test.
`SELECT INTO range FROM company.Person`			
`SELECT FROM select.Person`			
`SELECT FROM select`			
`SELECT FROM company.Person PARAMETERS int this`			
`SELECT FROM company.Person VARIABLES long this`			
A14.4-7: Keywords are permitted as field names only if they are on the right side of the "." in field access expressions	**valid:** `SELECT this.select FROM query.JDOQLKeywordsAsFieldNames`	`jdoql.keywords.KeywordsAsFieldNames`	Positive and negative test. New pc class required.
invalid: `SELECT select FROM query.JDOQLKeywordsAsFieldNames`			
A14.6.13-1: The String version of Query represents all query elements using a single string. The string contains the following structure:	`SELECT firstname AS firstName, lastname AS lastName INTO FullName FROM company.FullTimeEmployee WHERE salary > 1000 & projects.contains(project) & project.budget > limit VARIABLES Project project PARAMETERS BigDecimal limit IMPORTS IMPORT company.Project; IMPORT java.math.BigDecimal GROUP BY firstname, lastname HAVING lastname.startsWith('emp') ORDER BY personid RANGE 1 TO 5`	`jdoql.keywords.SingleString`	Positive test. New result class required.
A14.6.13-2: Keywords, identified above in **bold**, are either all upper-case or all lower-case. Keywords cannot be mixed case.	**valid:** `SELECT FROM company.Person`	`jdoql.keywords.UpperCaseLowercase`	Positive and negative test.
valid: `select from company.Person`			
valid: `select FROM company.Person`			
invalid: `SeLeCt FrOm company.Person`			

New Operators

Assertion	JDOQL	Testclass	Comment
A14.6.2-40: modulo operator	`SELECT FROM company.Person WHERE personid % 2 == 0`	`jdoql.operators.Modulo`	Positive test.
A14.6.2-41: instanceof operator	`SELECT FROM company.Employee WHERE mentor instanceof company.PartTimeEmployee`	`jdoql.operators.Instanceof`	Positive test.

New Supported Methods

Assertion	JDOQL	Testclass	Comment
A14.6.2-46: Supported Map methods: • `get(Object)` • `containsKey(Object)` • `containsValue(Object)`	**`get:`** `SELECT FROM Person WHERE phoneNumbers.get('home') == '1111'`	`jdoql.methods.SupportedMapMethods`	Positive Test.
`containsKey:` `SELECT FROM Person WHERE phoneNumbers.containsKey ('home')`			
`containsValue:` `SELECT FROM Person WHERE phoneNumbers.get('1111')`			
A14.6.2-47: New supported String methods: • `toLowerCase()` • `toUpperCase()` • `indexOf(String)` • `indexOf(String, int)` • `matches(String)` • `substring(int)` • `substring(int, int)` • `startsWith()` • `endsWith()`			
`toLowerCase:` `SELECT FROM company.Person WHERE firstname.toLowerCase() == 'john'`	`jdoql.methods.SupportedStringMethods`	Positive test.	
`toUpperCase:` `SELECT FROM company.Person WHERE firstname.toUpperCase() == 'EMP1FIRST'`			
`indexOf:` `SELECT FROM company.Person WHERE firstname.indexOf ('First') == 4`			
`indexOf:` `SELECT FROM company.Person WHERE firstname.indexOf ('First', 2) == 4`			
`matches:` `SELECT FROM company.Person WHERE firstname.matches ('*First')`			
`substring:` `SELECT FROM company.Person WHERE firstname.substring(4) == 'First'`			
`substring:` `SELECT FROM company.Person WHERE firstname.substring (4,9) == 'First'`			
`startsWith:` `SELECT FROM company.Person WHERE firstname.startsWith ('emp')`			
`endsWith:` `SELECT FROM company.Person WHERE firstname.endsWith ('First')`			
A14.6.2-48: Supported Math methods: • `Math.abs(numeric)` • `Math.sqrt(numeric)`	`SELECT FROM company.FullTimeEmployee WHERE Math.abs (salary) > 10000`	`jdoql.methods.SupportedMathMethods`	Positive test.
`SELECT FROM company.FullTimeEmployee WHERE Math.sqrt (salary) > 100`			
A14.6.2-49: Supported JDOHelper methods: • `JDOHelper.getObjectId(Object)`	`SELECT JDOHelper.getObjectId(this) FROM company.Person`	`jdoql.methods.SupportedJDOHelperMethods`	Positive test.

Parameters

Assertion	JDOQL	Testclass	Comment
A14.6.3-2: Parameters must all be declared explicitly via `declareParameters` or all be declared implicitly in the filter.	**valid:** `SELECT FROM company.Person WHERE firstname = param PARAMETERS String param`	`jdoql.parameters.MixedParameters`	Positive and negative test.
valid: `SELECT FROM company.Person WHERE firstname = :param`			
invalid: `SELECT FROM company.Person WHERE firstname = param`			
A14.6.3-3: Parameters implicitly declared (in the result, filter, grouping, ordering, or range) are identified by prepending a ":" to the parameter everywhere it appears. All parameter types can be determined by one of the following techniques:	**result:** `SELECT avg(employee.salary), :limit FROM company.FullTimeEmployee WHERE employee.salary > :limit`	`jdoql.parameters.ImplicitParameters`	Positive test.
filter: `SELECT FROM company.Person WHERE firstname = :param`			
grouping: `...`			
ordering: `...`			
range: `SELECT FROM company.FullTimeEmployee RANGE :one TO :ten`			
A14.6.13-3: If implicit parameters are used, their order of appearance in the query determines their order for binding to positional parameters for execution.	`SELECT FROM company.Person WHERE firstname == :param1 & lastname == :param2`	`jdoql.parameters.OrderOfParameters`	Positive test.

Variables

Assertion	JDOQL	Testclass	Comment
A14.6.5-1: ?? A variable that is not constrained with an explicit contains clause is constrained by the extent of the persistence capable class (including subclasses).	`SELECT department FROM company.Person WHERE firstname.endsWith ('First') VARIABLES Department department`	`jdoql.variables.UnconstrainedVariable`	Posistive test.
A14.6.5-2: ?? If the class does not manage an `Extent`, then no results will satisfy the query.	*`(javax.jdo.option.UnconstrainedQueryVariables)`* `SELECT v FROM Person VARIABLES NoExtent v`	`jdoql.variables.VariablesWithoutExtent`	Positive test. New pc class required.
A14.6.5-3: All variables must be explicitly declared, or all variables must be implicitly declared.	**explicit:** `SELECT FROM company.Employee WHERE team.contains(employee) & employee.firstname == 'emp1First' & projects.contains(project) & project.name == 'orange' VARIABLES Employee employee; Project project`	`jdoql.variables.MixedVariables`	Positive and negative test.
implicit: `SELECT FROM company.Employee WHERE team.contains(employee) & employee.firstname == 'emp1First' & projects.contains(project) & project.name == 'orange'`			
invalid: `SELECT FROM company.Company WHERE departments.contains (department) & department.name == 'Development' VARIABLES Employee employee`			
A14.6.5-4: Names are treated as variable names if they are explicitly declared via `declareVariables`. Otherwise, names are treated as field names if they are members of the candidate class. Finally, names are treated as implicitly defined variable names.	**explicit:** `SELECT FROM company.Employee WHERE team.contains(employee) & employee.firstname == 'emp1First' VARIABLES Employee employee`	`jdoql.variables.VariablesAndFields`	Positive test.
implicit: `SELECT FROM company.Employee WHERE team.contains(employee) & employee.firstname == 'emp1First'`			
field name: `SELECT FROM company.Person WHERE firstname == 'emp1First'`			

Other Language Changes

Assertion	JDOQL	Testclass	Comment
A14.6.2-42: There is no distinction made between character literals and `String` literals. Single character `String` literals can be used wherever character literals are permitted. `String` literals are allowed to be delimited by single quote marks or double quote marks. This allows `String` literal filters to use single quote marks instead of escaped double quote marks.	**valid:** `SELECT FROM mylib.PrimitiveTypes WHERE stringNull.startsWith('Even') OR charNotNull == 'O'`	`jdoql.CharacterAndStringLiterals`	Positive and negative test.
invalid: `SELECT FROM mylib.PrimitiveTypes WHERE stringNull.startsWith('Even') OR charNotNull == 'O.'`			
A14.6.2-43: Identifiers that are persistent field names or `public final static` field names are required to be supported by JDO implementations.	**field names:** `SELECT FROM company.Person VARIABLES String firstname PARAMETERS long personid`	`jdoql.IdentifiersEqualFieldNames`	Positive test.
static field names: `SELECT FROM fieldtypes.AllTypes WHERE NUM_VALUES == 10`			
A14.6.8-1: `setRange(long fromIncl, long toExcl)`	`SELECT lastname FROM company.Person RANGE 1 TO 10`	`jdoql.PositiveRange`	Positive test.
A14.6.8-2: If `((toExcl - fromIncl) <= 0)` evaluates to `true`, if the result of the query execution is a List, the returned List contains no instances, and an Iterator obtained from the List returns `false` to `hasNext()`. If the result of the query execution is a single instance (`setUnique(true)`), it will have a value of `null`.	`SELECT lastname FROM company.Person RANGE 10 TO 1`	`jdoql.NegativeRange`	Positive test.
A14.6.8-3: `setRange(String range);`	`SELECT lastname FROM company.Person RANGE 1 TO 10`	`jdoql.RangeAsString`	Positive test.

Query API Extensions

Assertion	JDOQL	Testclass	Comment
A14.5-11: Construct a new query instance using the specified `String` as the single-string representation of the query.	`SELECT FROM company.Person`	`api.NewQuerySingleString`	Positive test.
A14.5-12: Construct a new query instance with the given candidate class from a named query.	**valid, unique is false, unmodifiable is false:** `SELECT firstname INTO ...FullName FROM company.Person`	`api.NewNamedQuery`	Positive test and negative test. Add JDO metadata for named queries.
valid, unique is true, unmodifiable is false: `SELECT firstname INTO ...FullName FROM company.Person WHERE firstname == 'emp1First'`			
invalid, unique is true, unmodifiable is false: `SELECT firstname INTO ...FullName FROM company.Person`			
invalid, unique is false, unmodifiable is true: `SELECT firstname INTO ...FullName FROM company.Person`			
A14.5-13: If the named query is not found in already-loaded metadata, the query is searched for using an algorithm. Files containing metadata are examined in turn until the query is found. The order is based on the metadata search order for class metadata, but includes files named based on the query name.	`SELECT FROM company.Person`	`api.MetadataSearchOrder`	Positive test. Add JDO metadata for named queries.

A14.5-14: If the metadata is not found in the above, a `JDOUserException` is thrown.		`api.NamedQueryNotFound`	Negative test.
A14.5-15: The `Query` instance returned from this method can be modified by the application, just like any other `Query` instance.	`SELECT FROM company.Person WHERE firstname == 'emp1First'`	`api.ChangeQuery`	Positive test.
A14.5-16: Named queries must be compilable. Attempts to get a named query that cannot be compiled result in `JDOUserException`.	`SeLeCt FrOm company.Person`	`api.InvalidNamedQuery`	Negative test. Add JDO metadata for named queries.
A14.6-21: This method retrieves the fetch plan associated with the `Query`. It always returns the identical instance for the same `Query` instance. Any change made to the fetch plan affects subsequent query execution.	`SELECT FROM company.Person`	`api.FetchPan`	Positive test.
A14.6-16: `void setResult (String result);` Specify the results of the query if not instances of the candidate class.	**valid:** `SELECT lastname FROM company.Person`	`api.SetResult`	Positive test and negative test.
invalid: `SELECT noname FROM company.Person`			
A14.6-17: `void setGrouping (String grouping);` Specify the grouping of results for aggregates.	`SELECT lastname FROM company.Person GROUP BY lastname`	`api.SetGrouping`	Positive test.
A14.6-18: `void setUnique (boolean unique);` Specify that there is a single result of the query.	`SELECT UNIQUE firstname FROM company.Person WHERE lastname == emp1Last'`	`api.SetUnique`	Positive test.
A14.6-19: `void setResultClass (Class resultClass);` Specify the class to be used to return result instances.	`SELECT firstname, lastname INTO ...FullName FROM company.Person`	`api.SetResultClass`	Positive test. New result class required.
A14.6-20: `setRange(int fromIncl, int toExcl);` Specify the number of instances to skip over and the maximum number of result instances to return.	`SELECT FROM company.Person RANGE 1 TO 10`	`api.SetRange`	Positive test.
A14.6-22: The `Unmodifiable` option, when set to `true`, disallows further modification of the query, except for specifying the range and result class and `ignore Cache` option.	`SELECT FROM company.Person`	`api.UnmodifiableQuery`	Negative test.
A14.6-23: The single string query is first parsed to yield the result, result class, filter, variable list, parameter list, import list, grouping, ordering, and range. Then, the values specified in APIs `setResult`, `setResultClass`, `setFilter`, `declareVariables`, `declareParamters`, `declareImports`, `setGrouping`, `setOrdering`, and `setRange` override the corresponding settings from the single string query.	`SELECT firstname AS firstName, lastname AS lastName INTO ... FullName FROM company.FullTimeEmployee WHERE salary > 1000 & projects.contains(project) & project.budget > limit VARIABLES Project project PARAMETERS BigDecimal limit ORDER BY salary GROUP BY firstname, lastname HAVING lastname.startsWith('R') RANGE 1 TO 10`	`api.SingleStringQuery`	Positive test. New result class required.
A14.9-1: Some JDO vendors provide extensions to the query, and these extensions must be set in the query instance prior to execution.	`SELECT FROM company.Person`	`api.QueryExtentions`	Positive test.

Result Handling

Assertion	JDOQL	Testclass	Comment
A14.6.9-1: If `distinct` is specified, the query result does not include any duplicates. If the result parameter specifies more than one result expression, duplicates are those with matching values for each result expression.	`SELECT DISTINCT FROM company.Person`	`result.DistinctQuery`	Positive test.
A14.6.9-2: Queries against an extent always consider only distinct candidate instances, regardless of whether `distinct` is specified. Queries against a collection might contain duplicate candidate instances; the `distinct` keyword removes duplicates from the candidate collection in this case.	*`(javax.jdo.option.UnconstrainedQueryVariables)`* `SELECT FROM company.Person VARIABLES Project project`	`jdoql.DistintCandidateInstances`	Positive test.
A14.6.9-3: If a variable or a field of a variable is included in the result, either directly or via navigation through the variable, then the semantics of the contains clause that include the variable change. In this case, all values of the variable that satisfy the filter are included in the result.	**`variable:`** `SELECT project FROM company.Employee WHERE projects.contains(project) & project.name == 'orange' VARIABLES Project project`	`result.VariableInResult`	Positive test.
`field of variable:` `SELECT project.name FROM company.Employee WHERE projects.contains(project) & project.name == 'orange' VARIABLES Project project`			
A14.6.9-4: If any result is a navigational expression, and a non-terminal field or variable has a `null` value for a particular set of conditions (the result calculation would throw `NullPointerException`), then the result is null for that result expression.	**`field:`** `SELECT FROM company.Employee WHERE projects.contains(project)`	`result.NPEInResultExpr`	Positive test.
`variable:` `SELECT FROM company.Employee WHERE firstname == variable.firstname VARIABLES Employee variable;`			
A14.6.9-5: The result expressions include: ... The result expression can be explicitly cast using the (cast) operator.	`SELECT DISTINCT (FullTimeEmployee)manager FROM company.Employee`	`result.CastResult`	Positive test.
A14.6.9-6: `Count` returns `Long`. `Sum` returns `Long` for integral types and the field's type for other `Number` types (`BigDecimal`, `BigInteger`, `Float`, and `Double`). `Sum` is invalid if applied to non-`Number` types. `Avg`, `min`, and `max` return the type of the expression.	**`Count:`** `SELECT COUNT(salary) from company.FullTimeEmployee`	`result.AggregateResult`	Positive and negative test.
`Sum:` `SELECT SUM(salary) from company.FullTimeEmployee` *`(TBD for all integral types)`*			
`invalid Sum:` `SELECT SUM(hiredate) from FullTimeEmployee` *`(TBD for all non-Number types)`*			
`Avg:` `SELECT AVG(salary) from company.FullTimeEmployee` *`(TBD for all integral types)`*			
`invalid Avg:` `SELECT AVG(hiredate) from FullTimeEmployee` *`(TBD for all non-Number types)`*			
`Min:` `SELECT MIN(salary) from company.FullTimeEmployee` *`(TBD for all integral types)`*			
`Max:` `SELECT MAX(salary) from company.FullTimeEmployee` *`(TBD for all integral types)`*			
A14.6.9-7: If the returned value from a query specifying a result is `null`, this indicates that the expression specified as the result was `null`.	**`valid:`** `SELECT lastname FROM company.PERSON`	`result.NullResults`	Positive test.
A14.6.9-8: If not specified, the result defaults to `distinct this as C`	`SELECT FROM company.Department`	`result.DefaultResult`	Positive test.
A14.6.10-1: When grouping is specified, each result expression must be one of: an expression contained in the grouping expression; or, an aggregate expression evaluated once per group. The query groups all elements where all expressions specified in `setGrouping` have the same values. The query result consists of one element per group.	**`valid:`** `SELECT department, SUM(salary) FROM company.FullTimeEmployee GROUP BY department`	`result.Grouping`	Positive and negative test.
`invalid:` `SELECT department, salary FROM company.FullTimeEmployee GROUP BY department`			
A14.6.10-2: When `having` is specified, the `having` expression consists of arithmetic and boolean expressions containing aggregate expressions.	**`valid:`** `SELECT department, SUM(salary) FROM company.FullTimeEmployee GROUP BY department HAVING COUNT(department.employees) > 0`	`result.Having`	Positive and negative test.
`invalid:` `SELECT department, SUM(salary) FROM company.FullTimeEmployee GROUP BY department HAVING firstname == 'emp1First'`			
A14.6.11-1: When the value of the `Unique` flag is `true`, then the result of a query is a single value, with `null` used to indicate that none of the instances in the candidates satisfied the filter. If more than one instance satisfies the filter, and the range is not limited to one result, then `execute` throws a `JDOUserException`.	**`valid, result is non-null:`** `SELECT UNIQUE FROM company.Company WHERE companyid == 1`	`result.Unique`	Positive and negative test.
`valid, result is null:` `SELECT UNIQUE FROM company.Company WHERE name == 'non-existent'`			
`invalid:` `SELECT UNIQUE FROM company.Person`			

A14.6.11-2: The default Unique setting is `true` for aggregate results without a grouping expression, and `false` otherwise.	**`true`:** `SELECT COUNT(THIS) FROM company.Person`	`result.DefaultUnique`	Positive test.	
`false` grouping: `SELECT FROM company.Person GROUP BY lastname`				
`false`: `SELECT FROM company.Person`				

<ac:structured-macro ac:name="unmigrated-wiki-markup" ac:schema-version="1" ac:macro-id="ed9d5f5c-d20e-40c2-a25e-47235ca02ea6"><ac:plain-text-body><![CDATA[	*A14.6.12-1:*The result class may be one of the `java.lang` classes `Character`, `Boolea`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `String`, or `Object[]`; or one of the `java.math` classes `BigInteger` or `BigDecimal`; or the `java.util` class `Date`; or the `java.util` interface `Map`; or one of the `java.sql` classes `Date`, `Time`, or `Timestamp`; or a user-defined class.]]></ac:plain-text-body></ac:structured-macro>
	• If there are multiple result expressions, the result class must be able to hold all elements of the result specification or a `JDOUserException` is thrown.
	• If there is only one result expression, the result class must be assignable from the type of the result expression or must be able to hold all elements of the result specification. A single value must be able to be coerced into the specified result class (treating wrapper classes as equivalent to their unwrapped primitive types) or by matching. If the result class does not satisfy these conditions, a `JDOUserException` is thrown.
	• A constructor of a result class specified in the setResult method will be used if the results specification matches the parameters of the constructor by position and type. If more than one constructor satisfies the requirements, the JDO implementation chooses one of them. If no constructor satisfies the results requirements, or if the result class is specified via the `setResultClass` method, the following requirements apply:
	  ◦ A user-defined result class must have a no-args constructor and one or more public “set” or “put” methods or fields.
	  ◦ Each result expression must match one of:
	  ▪ a public field that matches the name of the result expression and is of the type (treating wrapper types the same as primitive types) of the result expression;
	  ▪ or if no public field matches the name and type, a public “set” method that returns `void` and matches the name of the result expression and takes a single parameter which is the exact type of the result expression;
	  ▪ or if neither of the above applies, a public method must be found with the signature `void put(Object, Object)` in which the first argument is the name of the result expression and the second argument is the value from the query result.
	  ◦ Portable result classes do not invoke any persistence behavior during their no-args constructor or set methods.
		valid, result class is String:
		`SELECT stringNull INTO String FROM mylib.PrimitiveTypes`
		`TBD for all supported JDK classes.`
			`result.ResultClassRequirements`	Positive and negative test. New result class required.	

valid, result class is TCK class: `SELECT stringNull AS s, intNotNull AS i INTO ... StringIntResult FROM mylib.PrimitiveTypes`				
invalid, result class is TCK class not having Long property: `SELECT stringNull AS s, longNotNull AS l INTO ... StringIntResult FROM mylib.PrimitiveTypes`				
invalid, result class is JDK class: `SELECT stringNull, intNotNull INTO String FROM mylib.PrimitiveTypes`				
invalid, result class is JDK class and not assignment compatible: `SELECT stringNull INTO Integer FROM mylib.PrimitiveTypes`				
invalid, result class is TCK class and not assignment compatible: `SELECT longNotNull AS s INTO ...StringIntResult FROM mylib.PrimitiveTypes`				
valid, specifying a constructor: `SELECT new StringIntResult(stringNull, intNotNull) FROM mylib.PrimitiveTypes`				
valid, specifying a non-existent constructor with AS: `SELECT new StringIntResult(stringNull AS s) FROM mylib.PrimitiveTypes`				
invalid, specifying a non-existent constructor without AS: `SELECT new StringIntResult(stringNull) FROM mylib.PrimitiveTypes`				
invalid, result class is TCK class not having an no-arg constructor: `SELECT stringNull INTO ...NoArgConstructor FROM mylib.PrimitiveTypes`				
invalid, result class is TCK class not having public fields and methods: `SELECT stringNull INTO ...NoFieldsNoMethods FROM mylib.PrimitiveTypes`				
valid, result class is TCK class having public fields and set methods: `SELECT stringNull AS s INTO ...FieldsAndSetMethods FROM mylib.PrimitiveTypes`				
valid, result class is TCK class having public fields and a put method: `SELECT stringNull AS s INTO ...FieldsAndPutMethod FROM mylib.PrimitiveTypes`				
valid, result class is TCK class having put method: `SELECT stringNull AS s INTO ...PutMethod FROM mylib.PrimitiveTypes`				
A14.6.12-2: Table 6: Shape of Result (C is the candidate class)	**valid:** `SELECT FROM company.Person`	`result.ShapeOfResult`	Positive test.	
valid, this as C: `SELECT this AS Person FROM company.Person`				
valid, unique: `SELECT UNIQUE FROM company.Person WHERE firstname == 'emp1First'`				
valid, unique, this as C: `SELECT UNIQUE this AS Person FROM company.Person WHERE firstname == 'emp1First'`				

valid, firstname: SELECT firstname FROM company.Person
valid, unique, firstname: SELECT UNIQUE firstname FROM company.Person WHERE firstname == 'emp1First'
valid, firstname, lastname: SELECT firstname, lastname FROM company.Person
valid, unique, firstname, lastname: SELECT UNIQUE firstname, lastname FROM company.Person WHERE firstname == 'emp1First'
valid, JDK result class, firstname: SELECT firstname INTO String FROM company.Person
valid, unique, JDK result class, firstname: SELECT UNIQUE firstname INTO String FROM company.Person WHERE firstname == 'emp1First'
valid, TCK result class: SELECT INTO ...PersonResult FROM company.Person
valid, unique, TCK result class: SELECT UNIQUE INTO ...PersonResult FROM company.Person WHERE firstname == 'emp1First'
valid, TCK result class, firstname: SELECT firstname INTO ...FullName FROM company.Person
valid, unique, TCK result class, firstname: SELECT UNIQUE firstname INTO ...FullName FROM company.Person WHERE firstname == 'emp1First'
valid, TCK result class, firstname, lastname: SELECT firstname, lastname INTO ...FullName FROM company.Person
valid, unique, TCK result class, firstname, lastname: SELECT UNIQUE firstname, lastname INTO ...FullName FROM company.Person WHERE firstname == 'emp1First'

SQL Queries

Assertion	JDOQL	Testclass	Comment
A14.7-1: In this case, the factory method that takes the language string and Object is used: `newQuery (String language, Object query)`. The language parameter is `javax.jdo.query.SQL` and the query parameter is the SQL query string.	SELECT PERSONID FROM persons	sql.NewQuery	Positive test.
A14.7-2: The only methods that can be used are `setClass` to establish the candidate class, `setUnique` to declare that there is only one result row, and `setResultClass` to establish the result class.	**setClass:** SELECT PERSONID FROM persons	sql.AllowedAPIMethods	Positive and negative test. New result class required.
setUnique(true): SELECT PERSONID FROM persons WHERE FIRSTNAME = 'emp1First'			
setResultClass(...FullName): SELECT FIRSTNAME AS firstName, lastname AS lastName FROM persons			
invalid: *for all other query api methods*			
parameter binding: SELECT PERSONID FROM persons WHERE FIRSTNAME = ?			
A14.7-3: SQL queries can be defined without a candidate class. These queries can be found by name using the factory method `newNamedQuery`, specifying the class as `null`, or can be constructed without a candidate class.	**named query:** SELECT PERSONID FROM persons	sql.CandidateClass	Positive test. Add JDO metadata for named SQL queries.
non-named query: SELECT PERSONID from persons			
A14.7-4: Table 7: Shape of Result of SQL Query	**valid, candidate class, unique is false:** SELECT PERSONID FROM persons	sql.ShapeOfResult	Positive test and negative test.
valid, candidate class, unique is true: SELECT PERSONID FROM persons WHERE FIRSTNAME = 'emp1First'			
invalid, candidate class, unique is true: SELECT PERSONID FROM persons			
valid, single column, unique is false: SELECT FIRSTNAME FROM persons			
valid, multiple columns, unique is false: SELECT FIRSTNAME, LASTNAME FROM persons			
valid, multiple columns, unique is true: SELECT FIRSTNAME, LASTNAME FROM persons WHERE FIRSTNAME = 'emp1First'			
valid, candidate class, result class, unique is false: SELECT FIRSTNAME AS firstName, LASTNAME AS lastName FROM persons			
valid, candidate class, result class, unique is true: SELECT FIRSTNAME AS firstName, LASTNAME AS lastName FROM persons WHERE FIRSTNAME = 'emp1First'			

valid, result class, unique is false: `SELECT FIRSTNAME AS firstName, LASTNAME AS lastName FROM persons`
valid, result class, unique is true: `SELECT FIRSTNAME AS firstName, LASTNAME AS lastName FROM persons WHERE FIRSTNAME = 'emp1First'`
valid, result class (binding using put method), unique is false: `SELECT FIRSTNAME, LASTNAME FROM persons`
valid, result class (binding using put method), unique is true: `SELECT FIRSTNAME, LASTNAME FROM persons WHERE FIRSTNAME = 'emp1First'`

Deletion by Query

Assertion	JDOQL	Test class	Comment
A14.8-1: These methods delete the instances of affected classes that pass the filter, and all dependent instances. Affected classes are the candidate class and its persistence-capable subclasses.	`SELECT * FROM company.Person`	`delete.DeletePersistentAll`	Positive test.
A14.8-2: The number of instances of affected classes that were deleted is returned. Embedded instances and dependent instances are not counted in the return value.	`SELECT * FROM company.Person`	`delete.NoOfDeletedInstances`	Positive test.
A14.8-3: Query elements `filter`, `parameters`, `imports`, `variables`, and `unique` are valid in queries used for delete. Elements `result`, `result class`, `range`, `grouping`, and `ordering` are invalid. If any of these elements is set to its non-default value when one of the `deletePersistentAll` methods is called, a `JDOUserException` is thrown and no instances are deleted.	**valid:** `SELECT UNIQUE FROM company.FullTimeEmployee WHERE salary > 1000 & projects.contains(project) & project.budget > limit VARIABLES Project project PARAMETERS BigDecimal limit`	`delete.DeleteQueryElements`	Positive and negative test. New result class required.
invalid result: `SELECT firstname AS firstName INTO ...FullName FROM company.FullTimeEmployee`			
invalid order by: `SELECT FROM company.FullTimeEmployee ORDER BY salary`			
invalid group by: `SELECT FROM company.FullTimeEmployee GROUP BY lastname`			
invalid having: `SELECT FROM company.FullTimeEmployee GROUP BY lastname HAVING lastname.startsWith('R')`			
invalid range: `SELECT FROM company.FullTimeEmployee RANGE 1 TO 10`			

A14.8-4: Dirty instances of affected classes are first flushed to the datastore. Instances already in the cache when deleted via these methods or brought into the cache as a result of these methods undergo the life cycle transitions as if `deletePersistent` had been called on them. That is, if an affected class implements the `DeleteCallback` interface, the instances to be deleted are instantiated in memory and the `jdoPreDelete` method is called prior to deleting the instance in the datastore. If any `LifecycleListener` instances are registered with affected classes, these listeners are called for each deleted instance. Before returning control to the application, instances of affected classes in the cache are refreshed by the implementation so their status in the cache reflects whether they were deleted from the datastore.	`SELECT * FROM company.Person`	`delete.DeleteCallback`	Positive test.

Testcase Pattern

Positive test

A positive test expects that the query compiles and executes w/o exception and returns the expected result:

```
    Query query = pm.newQuery();
    ...
    // define query
    Object results = query.execute(...);

    // check query result
    List expected = new ArrayList();
    expected.add(...);
    checkQueryResultWithoutOrder(assertion, results, expected);
```

Negative test

A negative test case uses an invalid JDOQL query and expects an exception to be thrown by compile or execute:

```
    try {
        Query query = pm.newQuery();
        ...
        // define query
        Object results = query.execute();
        fail(ASSERTION_FAILED, text);
    } catch (JDOException e) {
        if (debug) logger.debug("Caught expected " + e);
    }
```