J

J2EE

► Java Enterprise Edition (JEE)

Java Annotations

► Java Metadata Facility

Java Application Server

► Application Server

Java Database Connectivity

Changqing Li
Duke University, Durham, NC, USA

Synonyms

JDBC

Definition

Java Database Connectivity (JDBC) [1] is an Application Programming Interface (API) for the Java programming language that enables Java programs to execute Structured Query Language (SQL) statements and defines how an application accesses a database. JDBC provides methods to query and update data in a database. Different from ODBC (Open Database Connectivity), JDBC is oriented towards relational databases only.

Key Points

JDBC (pronounced as separate letters) is similar to ODBC, but it is designed specifically for Java programs, whereas ODBC is language-independent. The Java Standard Edition includes the JDBC API as well as an ODBC implementation of the API enabling connections to any relational database that supports ODBC [1].

JavaSoft, a subsidiary of Sun Microsystems, developed JDBC. Since the release of Java Development Kit (JDK) 1.1, JDBC has been part of the Java Standard Edition. JDBC has been developed under the Java Community Process since version 3.0. JDBC 3.0 (included in Java 2 Standard Edition (J2SE) 1.4) is specified by Java Specification Request (JSR) 54, the JDBC Rowset additions are specified by JSR 114, and JDBC 4.0 (included in Java Standard Edition 6) is specified by JSR 221.

Multiple implementations of JDBC can exist and can be used by the same application. A mechanism is provided by the API to dynamically load the correct Java packages and register them with the JDBC Driver Manager, a connection factory for creating JDBC connections.

Creating and executing statements are supported by JDBC connections. These statements may either be update statements such as SQL CREATE, INSERT, UPDATE and DELETE or be query statements using the SELECT statement. In addition, a statement may invoke a stored procedure.

Because Java itself runs on most platforms, and since nearly all relational database management systems (DBMSs) support SQL, JDBC makes it possible to write a single database application that can run across different DBMSs on different platforms.

Cross-references

► Data Integration
► Database Adapter and Connector
► Interface
► .NET Remoting
► Open Database Connectivity
► Web 2.0/3.0
► Web Services

Recommended Reading

1. Hamilton G., Cattell R., and Fisher M. JDBC Database Access with Java: A Tutorial and Annotated Reference. Addison Wesley, Boston, MA, USA, 1997.

Java EE

► Java Enterprise Edition (JEE)

Java Enterprise Edition

Ricardo Jimenez-Peris, Marta Patiño-Martinez
Universidad Politecnica de Madrid, Madrid, Spain

Synonyms

J2EE; Java EE; JEE

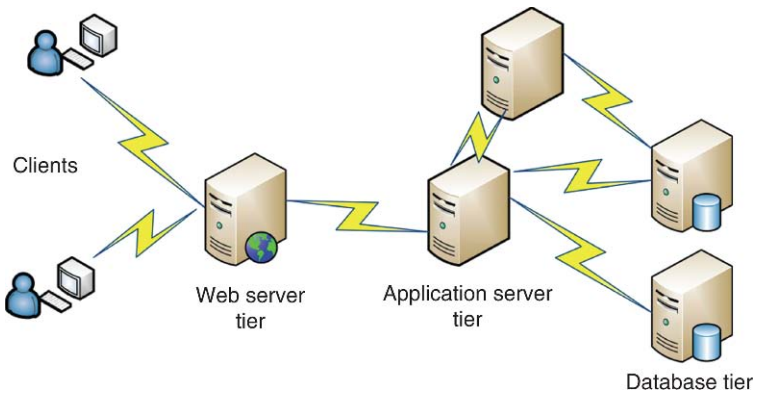
Definition

Java EE (JEE) is a Java Community Process (JCP) specification for Java application servers. JEE consists of a set of related specifications and APIs that shapes an ecosystem for building Java distributed applications over a multi-tier architecture, including web, business logic, and storage tiers. JEE provides components specific for each tier. JEE provides servlets and Java Server Pages (JSPs) for the web tier, Enterprise Java Beans (EJBs) for the business logic tier, and a data driver (JDBC) and an object-relational mapping (via entity beans) for the storage tier. Components can be distributed at different sites and interact with each other. The interaction among distributed JEE components is achieved via RMI (Remote Method Invocation), the JEE specification for remote procedure calls. In its latest editions, JEE has been smoothly integrated with XML processing and web services. It is possible to program web services using EJBs and a large set of APIs enable a standard way to effectively manipulate XML. JEE covers all important non-functional aspects such as transactions, security, etc. Transactions are core to JEE and are considered across all tiers in a consistent fashion through the Java Transaction Service and API (JTS/JTA). A number of specifications address various security issues such as authentication (JAAS, Java Authentication and Authorization Service), encryption, and so on.

Key Points

The JEE project originated back in 1998 when the JPE (the original name) initiative was announced by Sun Microsystems. J2EE 1.2 was released in Dec. 1999. The following versions have been released almost every two years, J2EE 1.3 in Sept. 2001 (1.3 was the first version to be developed as a JCP specification), J2EE 1.4 in Nov. 2003, JEE 1.5 (with version 1.5 J2EE was renamed as JEE) in May 2006, and JEE 1.6 is expected to be released sometime in 2008.

JEE is a multi-tier middleware framework to program all kinds of applications based on the Java language. Multi-tier architectures have become widely used since they allow an adequate separation of concerns. Each tier has a specific container with a component model tailored to the specific mission of the tier. Some of the most common tiers are: web tier, business logic tier, and persistent storage tier.



Java Enterprise Edition. Figure 1. JEE multi-tier architecture.

Figure 1 depicts a typical JEE multi-tier architecture. Clients interact with the system via the web server tier. The web server tier then delegates the business logic to the application server tier. The JEE application server tier can consist of multiple servers, that is, be distributed. Then, the persistent data are kept in the database tier. The data can be kept in multiple distributed databases. The database instances can be shared across application servers. Transactions are used to enforce data consistency across the entire system despite concurrent accesses to the data. Transactions also protect data against server failures providing failure atomicity.

For the web tier, JEE provides a component model for generating dynamic web content. This component model is based on Java Server Pages (JSPs) and servlets. JSPs provide a high level abstraction for dynamic content, whilst servlets provide a lower level view. In fact, JSPs are translated into servlets. Servlet containers are specialized web servers that support servlet execution. Typically servlets deal with the presentation aspects of an application and they are specialized to render dynamic web contents to users. Servlets delegate the business logic to the application server tier, that is, to the JEE application server.

The JEE application server provides a component model for writing general applications. The components in the JEE component model are termed Enterprise Java Beans (EJBs). Two different kinds of EJBs are distinguished: session beans and entity beans. Session beans enable to encapsulate the interaction with users. There are two flavors: stateless and stateful session beans. Stateless session beans provide support for non-conversational interactions, whilst stateful session beans support conversational interactions. Stateless session beans are instantiated per client invocation and their state is discarded after processing the client request that triggered their instantiation. On the other hand, stateful session beans are created upon the first client invocation and then, they are kept during all the client conversation. Their state enables to tracking of the status of the interaction across invocations (e.g., a shopping cart). From JEE 1.5, it is also possible to persist POJOs (Plain Old Java Object, that is, a Java Object) using JPA (Java Persistent API) without the need of a JEE application server.

JEE provides two models of interaction with the database: explicit and implicit. In the explicit model of interaction the EJBs access the database directly via a JDBC driver. The JDBC driver abstracts the database specifics of the underlying implementation providing a uniform vendor-independent interface. However, the object oriented paradigm has a different style than the relational model (the well-known object-relational impedance mismatch) that forces programmers to write a lot of repetitive code prone to errors to create queries and to transform query results into objects. The object-relational mapping (ORM) approach addresses this impedance mismatch by offering the programmer a view of an object oriented database, eliminating the need to write code.

The database tier access can be explicit via JDBC commands or implicit by using the object-relation mapping (ORM) of JEE. Explicit access implies issuing JDBC commands with SQL statements and dealing directly with the results sets returned by JDBC. Implicit access via ORM enables access to the persistent state using the object-oriented paradigm. Tuples are modeled as special kind of JEE components, entity beans, that are accessed as regular objects. Entity beans provide an object-oriented cache of the database. The JEE system takes care of updating the database and implementing the necessary concurrency control to provide transactional serializability.

JEE also provides support for the publish-subscribe paradigm. For that purpose it provides a specific component model and the associated container, Java Message Service (JMS). JMS provides message-driven beans that are components that are activated upon the reception of a particular kind of message. Message-driven beans can be invoked by client applications and EJBs and they can also invoke other EJBs. JMS provides an asynchronous interaction model that complements the synchronous one provided by EJBs and RMI.

In the last few years, a number of initiatives have tried to simplify JEE development. One of the initiatives with highest impact has been the Spring framework [2]. Spring provides a lightweight approach to JEE simplifying the development of JEE applications. Spring is based heavily on programming interfaces (as opposed to programming objects) and the use of aspect oriented programming. Spring has inspired some of the major changes in JEE 1.5 and 1.6.

Cross-references

► Replication in Multi-Tier Architectures

Recommended Reading

1. JSR 316: Java™ Platform, Enterprise Edition 6 (Java EE 6) Specification. http://jcp.org/en/jsr/detail?id=316
2. Spring Framework. http://springframework.org/

Java Metadata Facility

David Buttler
Lawrence Livermore National Laboratory, Livermore, CA, USA

Synonyms

Java metadata facility; JSR 175; Java annotations

Definition

The Java Metadata Facility is introduced by Java Specification Request (JSR) 175 [1], and incorporated into the Java language specification [2] in version 1.5 of the language. The specification allows annotations on Java program elements: classes, interfaces, methods, and fields. Annotations give programmers a uniform way to add metadata to program elements that can be used by code checkers, code generators, or other compile-time or runtime components.

Annotations are defined by annotation types. These are defined the same way as interfaces, but with the symbol "@" preceding the "interface" keyword. There are additional restrictions on defining annotation types:

1. They cannot be generic.
2. They cannot extend other annotation types or interfaces.
3. Methods cannot have any parameters.
4. Methods cannot have type parameters.
5. Methods cannot throw exceptions.
6. The return type of methods of an annotation type must be a primitive, a String, a Class, an annotation type, or an array, where the type of the array is restricted to one of the four allowed types.

See [2] for additional restrictions and syntax.

The methods of an annotation type define the elements that may be used to parameterize the annotation in code. Annotation types may have default values for any of its elements. For example, an annotation that specifies a defect report could initialize an element defining the defect outcome to "submitted." Annotations may also have zero elements. This could be used to indicate serializability for a class (as opposed to the current Serializability interface).

Key Points

There are several annotation types that are predefined in the Java 1.5 programming language: "@Override," "@Deprecated," and "@SuppressWarnings" are the most common ones.

"@Override" indicates that a method in a subclass overrides a method from its superclass, as opposed to overloading it. This is an example of an annotation with zero elements. A common, yet difficult to identify, error in writing Java classes occurs when a programmer overloads the `equals` method, rather than overriding it. This leads to errors that are difficult to track down.

"@Deprecated" indicates that a class or method has been deprecated and that programmers should use an alternative. This replaces the javadoc "@deprecated" tag that served the same purpose.

"@SuppressWarnings" indicates that a compiler should not report warnings of a particular type. This particular annotation requires an element, such as "@SupressWarnings('unchecked')," defining the type of warning to ignore for the annotated compilation unit. Warning types are defined by the compiler and are not specified in the Java language specification.

Cross-references

▶ Metadata

Recommended Reading

1. Coward D. JSR 175: A Metadata Facility for the Java™ Programming Language, 2004. http://jcp.org/en/jsr/detail?id=175
2. Gosling J., Joy B., Steele G., and Bracha G. The Java™ Language Specification. Prentice Hall, NJ, USA, 2005.

JD

▶ Join Dependency

JDBC

▶ Java Database Connectivity

Join

Cristina Sirangelo
University of Edinburgh, Edinburgh, Scotland, UK

Definition

The join is a binary operator of the relational algebra that combines tuples of different relations based on a

relationship between values of their attributes. The primitive version of the join operator is called *natural join*. Given two relation instances R_1, over set of attributes U_1, and R_2 over set of attributes U_2, the natural join $R_1 \bowtie R_2$ returns a new relation, over set of attributes $U_1 \cup U_2$, consisting of tuples $\{t | t(U_1) \in R_1$ and $t(U_2) \in R_2\}$. Here $t(U)$ denotes the restriction of the tuple t to attributes in the set U.

A derivable version of the join operator is obtained by composing the natural join with the selection operator σ: the *theta-join* $R_1 \bowtie_\theta R_2$ is defined as $\sigma_\theta(R_1 \bowtie R_2)$, where θ is an arbitrary condition allowed in a generalized selection over set of attributes $U_1 \cup U_2$. In the case that θ is a conjunction of equality atoms of the form $A = B$, where A is an attribute in U_1 and B an attribute in U_2, the theta-join is called *equijoin*.

Another derivable join operator is the *semijoin*, denoted by $R_1 \ltimes R_2$; it is defined as $\pi_{U_1}(R_1 \bowtie R_2)$, where π_{U_1} denotes the projection on attributes U_1.

Key Points

In the natural join $R_1 \bowtie R_2$, tuples of R_1 and R_2 having the same values of common attributes are combined. If the sets of attributes of R_1 and R_2 are disjoint, $R_1 \bowtie R_2$ coincides with the cartesian product.

The natural join is often used to combine tuples based on attributes correlated by a foreign key-constraint: consider a relation *Students* over attributes (*student-number, student-name*), containing tuples {(1001, *Black*), (1002, *White*)}, and a relation *Exams* over attributes (*course-number, student-number, grade*), containing tuples {(*EH*1, 1001, *A*), (*EH*1, 1002, *A*), (*GH*5, 1001, *C*)}. Then the natural join *Students* $\bowtie$ *Exams* is a relation over attributes (*student-number, student-name, course-number, grade*) with tuples {(1001, *Black*, *EH*1, *A*), (1001, *Black*, *GH*5, *C*), (1002, *White*, *EH*1, *A*)}.

In the absence of attribute names the only primitive notion of join is the cartesian product. In this case operators of theta-join and equijoin can be derived by composing selection and cartesian product. More precisely, if θ is a boolean combination of atoms of the form $j \alpha k$ with $j \leq arity(R_1)$ and $k \leq arity(R_2)$ and $\alpha \in \{=, \neq, <, >, \leq, \geq\}$, then the theta-join $R_1 \bowtie_\theta R_2$ in the unnamed algebra is defined as $\sigma_{\theta'}(R_1 \times R_2)$, where θ' is obtained from θ by replacing each atom $j \alpha k$ with $j \alpha (arity(R_1) + k)$.

In each of the join operators described above (except the semijoin) there can be tuples of the input relations which do not occur in the output, because they satisfy the join condition with no tuple of the other relation. The *left (right) outer join* adds to the join of R_1 and R_2 all tuples of R_1 (R_2) not occurring in the join, completed with nulls on attributes of R_2 (R_1). The *full outer join* adds both tuples of R_1 and R_2 to the join.

Cross-references

- ▶ Cartesian Product
- ▶ Foreign Key
- ▶ Projection
- ▶ Relation
- ▶ Relational Algebra
- ▶ Selection

Join Dependency

Solmaz Kolahi
University of British Columbia, Vancouver, BC, Canada

Synonyms

JD

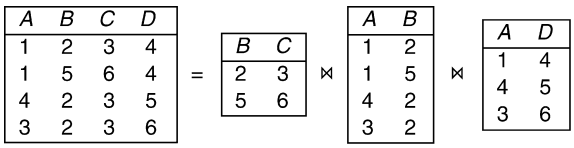
Definition

A *join dependency (JD)* over a relation schema $R[U]$ is an expression of the form $\bowtie [X_1,...,X_n]$, where $X_1 \cup \ldots \cup X_n = U$. An instance I of $R[U]$ satisfies $\bowtie [X_1,...,X_n]$ if $I = \pi_{X_1}(I) \bowtie \ldots \bowtie \pi_{X_n}(I)$. In other words, an instance satisfies the join dependency if it is equal to the join of its projections on the sets of attributes $X_1,...,X_n$. A multivalued dependency $X \twoheadrightarrow Y$ is a special case of a join dependency on two sets, and can be expressed as $\bowtie [XY, X(U - XY)]$, where XY represents $X \cup Y$.

Key Points

Join dependencies are particularly important in connection with the decomposition technique for schema design and normalization. The main goal of the decomposition technique is to avoid redundancies due to data dependencies by decomposing a relation into smaller parts. A good decomposition should have the *lossless join* property, meaning that no information should be lost after the decomposition. In other words, the original database instance should be retrievable by joining the smaller relations, and this can be expressed

by a JD. The following figure shows an instance of the relation schema $R[A, B, C, D]$ that satisfies the join dependency $\bowtie [BC, AB, AD]$:



Join dependencies are usually considered together with functional and multivalued dependencies (FDs and MVDs) in normalization. The implication problem of a JD from a set of JDs, MVDs, and FDs is known to be NP-hard. In addition, the implication problem of JDs cannot be axiomatized. That is, there is no sound and complete set of rules that can be used to check whether a dependency is implied by a set of JDs. However, there is a powerful tool, called *chase*, that could be used to reason about these dependencies in exponential time and space [1].

Cross-references

- ▶ Functional Dependency
- ▶ Join
- ▶ Multivalued Dependency
- ▶ Normal Forms and Normalization
- ▶ Projection

Recommended Reading

1. Abiteboul S., Hull R., and Vianu V. Foundations of Databases. Addison-Wesley, Reading, MA, 1995.

Join Index

Theodore Johnson
AT&T Labs – Research, Florham Park, NJ, USA

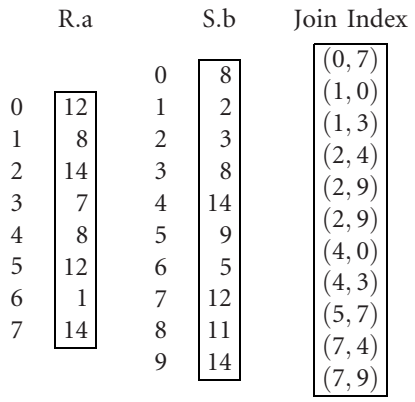
Definition

A join index is a collection of pairs $\{(r, s)\}$ such that the record in table R with record ID (RID) r joins with the record in table S with RID s, according to the *join* predicate which defines the index.

Key Points

The purpose of a join index is to accelerate common joins, even equijoins. One of the advantages of join indices is that they can be represented in a very compact way, allowing for highly efficient access. For example, suppose that the DBMS is to evaluate a query, "Select R.a from R,S where R.a = S.b." A conventional join would use a nested loop algorithm, with an indexed scan in the inner loop. With a join index, the join can be computed by scanning the join index, thus minimizing random I/O.

There are a variety of ways of implementing a join index. One can list pairs (clustered on R or clustered on S), or in the case of an equijoin, associate R and S RIDs with attribute values. The example below shows the join index for R.a = S.b, organized as a list of pairs.



Cross-references

- ▶ Join
- ▶ Star Index

Recommended Reading

1. Li Z. and Ross K.A. Fast joins using join indices. VLDB J., 8(1):1–24, 1999.
2. Valduriez P. Join indices. ACM Trans. Database Syst., 12(2):218–246, 1987.

Join Indices

- ▶ Star Index

Join Order

Jingren Zhou
Microsoft Research, Redmond, WA, USA

Synonyms

Join order; Join sequence

Definition

A database query typically contains multiple joins. When joins (for example, inner joins) are commutative and/or associative, there can more than one evaluation order for joins. The join order has an enormous impact on the query cost. One of the main responsibilities of the query optimizer is to determine the optimal join order for query evaluation.

Key Points

Choosing a good join order is very important to achieve a good query performance. One important consideration for choosing an join order is to reduce the size of intermediate results as much as possible. For example, it is beneficial to first evaluate a join that returns the least result. Other considerations include join methods, data properties, and access methods, etc.

Depending on the choice of join orders, query plans can be of different shapes.

- *Left-Deep* query plans use a base table as the inner table for each join.
- *Right-Deep* query plans use a base table as the outer table for each join.
- *Bushy* query plans use the intermediate result from other joins as join inputs.

Query optimizers typically use dynamic programming and heuristics to determine join orders.

Cross-references

▶ Parallel Join Algorithms
▶ Query Optimization
▶ Evaluation of Relational Operators

Recommended Reading

1. Mishra P. and Eich M.H. Join processing in relational databases. ACM Comput. Surv., 24(1):63–113, 1992.
2. Selinger P.G., Astrahan M.M., Chamberlin D.D., Lorie R.A., and Price T.G. Access path selection in a Relational Database Management System. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1979, pp. 23–34.

Join Processing

▶ Distributed Join

Join Sequence

▶ Join Order

JSR 175

▶ Java Metadata Facility