Business Components Development II

J2EE

JAVA INSTITUTE for Advanced Technology
INNOVATIONS FOR THE KNOWLEDGE SOCIETY
SGS ISO 9001:2008 Certified Institute
+94 11 250 6000
+94 71 862 3153
No: 3B 1/4, Havelock Road
Colombo 05, Sri Lanka
WTRC | KTRC | NTNRC Neuroscience Research

In this module we will discuss,

- Transaction Management
- Timer Service
- Web Services
- Interceptor
- JavaMail API
- Security
- EJB Exception Handling
- Package and Deploy EJB Applications

JNDI - Java Naming and Directory Interface

Java Naming and Directory Interface (JNDI) is a Java API that provides naming and directory functionality to Java applications.

It allows Java software clients to discover and look up data and objects via a name-based interface.

JNDI is particularly useful in enterprise applications where resources such as databases, messaging systems, and directory services need to be located dynamically.

Key features of JNDI include:

Naming Service: JNDI provides a unified interface for accessing different naming and directory services such as LDAP (Lightweight Directory Access Protocol), DNS (Domain Name System), and RMI (Remote Method Invocation).

Directory Service: JNDI allows Java applications to interact with directory services like LDAP, enabling tasks such as searching, adding, modifying, and deleting entries in a directory.

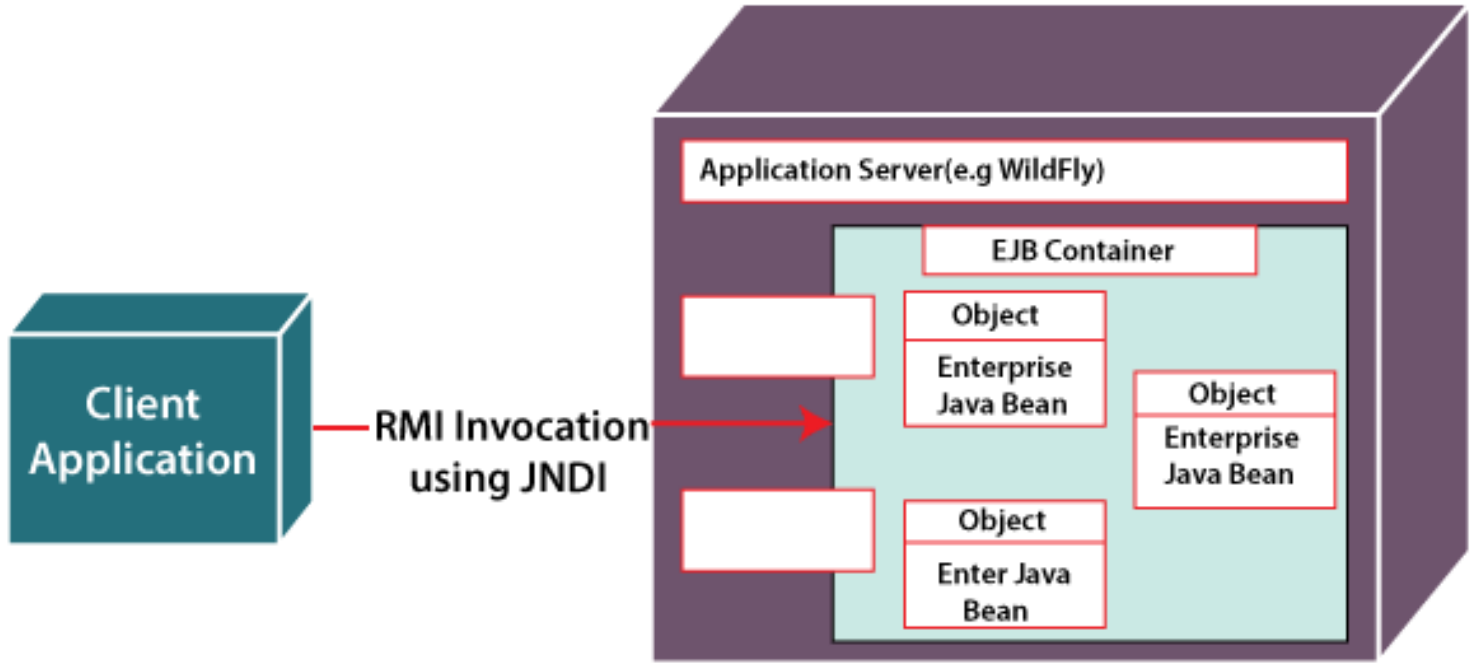
Resource Lookup: Applications can use JNDI to look up resources such as JDBC (Java Database Connectivity) data sources, JMS (Java Message Service) connection factories, and JavaMail sessions.

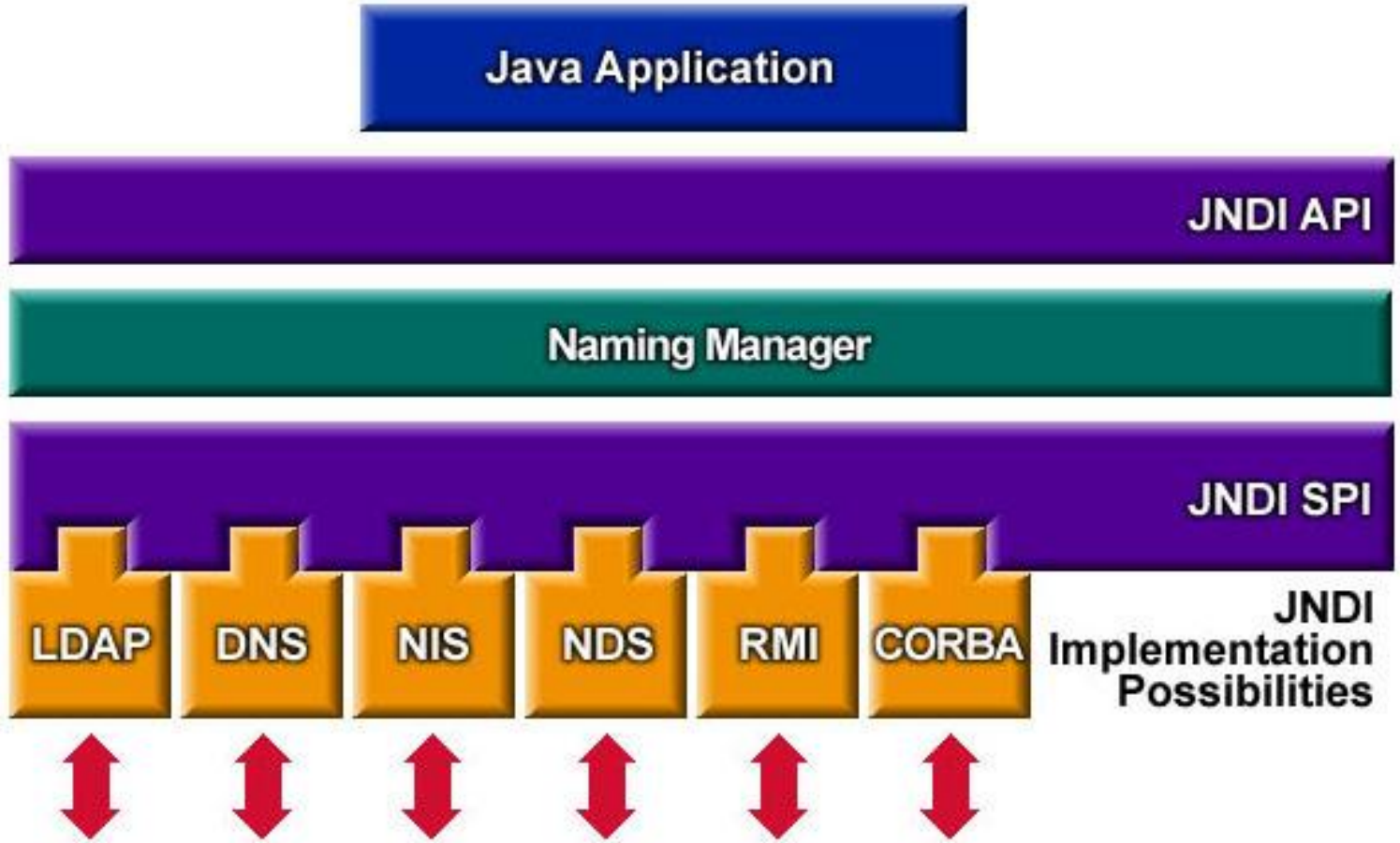
Key features of JNDI include:

Context Management: JNDI organizes resources in a hierarchical structure called a naming context. Contexts can be nested, allowing for a logical organization of resources.

Provider-Based Architecture: JNDI supports a provider-based architecture, allowing different providers to implement the naming and directory services. For example, there are different providers for LDAP, RMI, and DNS.

Integration with Java EE: JNDI is heavily used in Java Enterprise Edition (Java EE) applications for resource management, such as accessing database connections, JMS resources, and EJB (Enterprise JavaBeans) components.


Client Application
RMI Invocation using JNDI
Application Server(e.g WildFly)
EJB Container
Object
Enterprise Java Bean
Object
Enterprise Java Bean
Object
Enter Java Bean



Java Persistence API (JPA)

The Java ORM standard for storing, accessing, and managing Java objects in a relational database

JPA Introduction

The **Java Persistence API (JPA)** is a specification of Java. It is used to persist data between Java object and relational database. JPA acts as a bridge between object-oriented domain models and relational database systems.

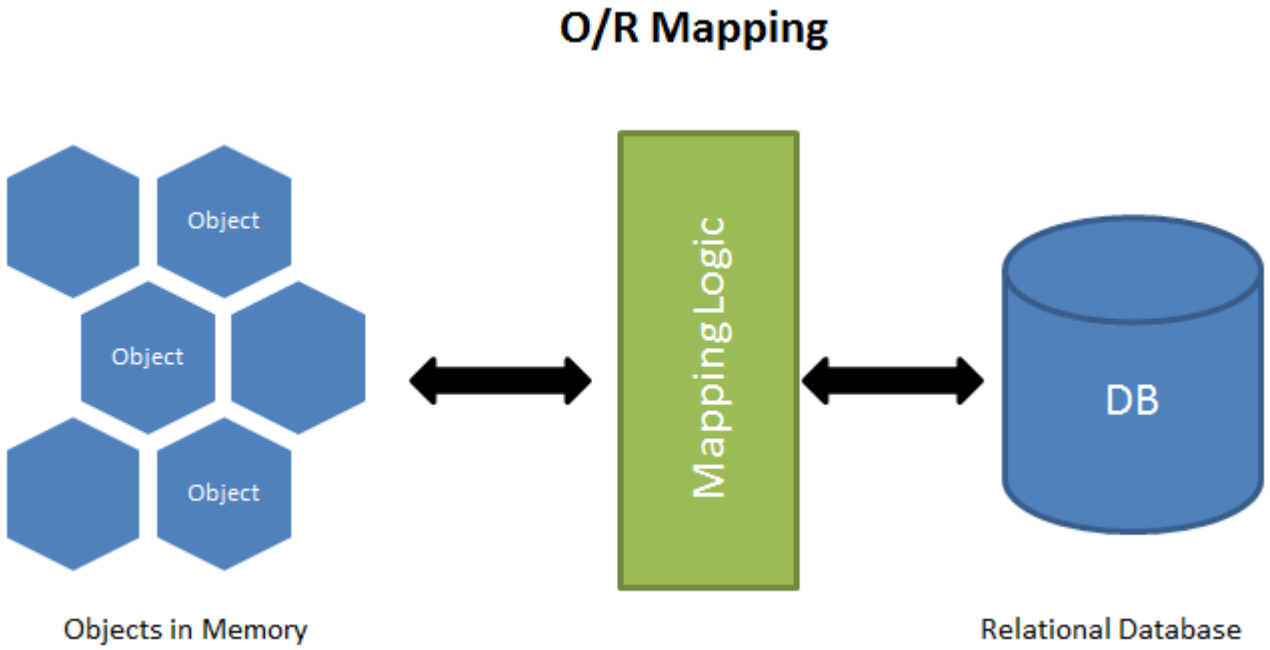
As JPA is just a specification, it doesn't perform any operation by itself. It requires an implementation. So, **ORM** tools like **Hibernate**, **TopLink** and **iBatis** implements JPA specifications for data persistence.

JPA Versions

- The first version of Java Persistenece API, **JPA 1.0** was released in 2006.
- **JPA 2.0**: Introduced in 2009, built on the Apache OpenJPA project, providing a standard for object-relational mapping in Java.
- **JPA 2.1**: Released in 2013, added features like the Criteria API enhancements and support for stored procedures.
- **JPA 2.2**: Launched in 2017, included improvements such as support for new data types and enhanced query capabilities.
- **Jakarta Persistence 3.0**: Released in 2021, this version marked the transition from Java EE to Jakarta EE, with updates to align with the new Jakarta namespace.
- **Jakarta Persistence 3.1**: Introduced in 2022, it brought further enhancements and optimizations.
- **Jakarta Persistence 3.2**: Released in 2023, this version included additional features and improvements to the API.
- **Jakarta Persistence 4.0**: Currently under development, expected to be released with Jakarta EE 12, aiming to introduce new features and improvements to the persistence model.

JPA Object Relational Mapping

Object Relational Mapping (ORM) is a functionality which is used to develop and maintain a relationship between an object and relational database by mapping an object state to database column. It is capable to handle various database operations easily such as inserting, updating, deleting etc.



ORM Frameworks

Following are the various frameworks that function on ORM mechanism: -

- Hibernate
- TopLink
- ORMLite
- iBATIS
- JPOX

Mapping Directions

Mapping Directions are divided into two parts: -

- **Unidirectional relationship -** In this relationship, only one entity can refer the properties to another. It contains only one owing side that specifies how an update can be made in the database.

- **Bidirectional relationship -** This relationship contains an owning side as well as an inverse side. So here every entity has a relationship field or refer the property to other entity.

Types of Mapping

Following are the various ORM mappings: -

- One-to-one
- One-to-many
- Many-to-one
- Many-to-many

JPA Entity Introduction

Entity Properties :-

- Persistability
- Persistent Identity
- Transactionality
- Granuality

Persistability

Persistability refers to the ability of an object to be stored in a persistent storage medium, such as a database, so that its state can be maintained beyond the lifecycle of the application. In JPA, entities are typically marked as persistent, meaning their state can be saved and retrieved from the database.

Persistent Identity

Persistent identity is the concept that an entity has a unique identifier that remains consistent across different states of the application. This identifier is typically represented by a primary key in the database.

Transactionality

Transactionality refers to the ability to execute a series of operations as a single unit of work, ensuring that either all operations succeed or none do. This is crucial for maintaining data integrity.

Granularity

Granularity in the context of persistence refers to the level of detail at which data is stored and managed. It can pertain to how fine or coarse the data model is, affecting performance and complexity.

Entity Metadata

Each entity is associated with some metadata that represents the information of it. Instead of database, this metadata is exist either inside or outside the class.

This metadata can be in following forms: -

- Annotation
- XML

JPA Creating an Entity

A Java class can be easily transformed into an entity. For transformation the basic requirements are: -

- No-argument Constructor
- XML or Annotation

JPA Entity Manager

Following are some of the important roles of an entity manager: -

- The entity manager implements the API and encapsulates all of them within a single interface.
- Entity manager is used to read, delete and write an entity.
- An object referenced by an entity is managed by entity manager.

Entity Manager Factory

When the application has finished using the entity manager factory, and/or at application shutdown, the application should close the **EntityManagerFactory**.

Once an **EntityManagerFactory** has been closed, all its entity managers are considered to be in the closed state.

The **EntityManagerFactory** interface present in **java.persistence** package is used to provide an entity manager.

Persistence Units

Persistence units are defined by the **persistence.xml** configuration file. The JAR file or directory whose **META-INF** directory contains **persistence.xml** is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit's root.

persistence.xml

```
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
  <persistence-unit name="AppPU" transaction-type="JTA">
    <jta-data-source>j2ee_jdbc_example</jta-data-source>
                    // OR
    <properties>
      <property name="jakarta.persistence.schema-generation.database.action" value="create"/>
      <property name="jakarta.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>
      <property name="jakarta.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/exampledb"/>
      <property name="jakarta.persistence.jdbc.user" value="username"/>
      <property name="jakarta.persistence.jdbc.password" value="password"/>
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
      <!-- Additional properties can be added as required -->
    </properties>
  </persistence-unit>
</persistence>
```

Steps to persist an entity object.

- Persistence
- createEntityManagerFactory() method
- EntityManager
- getTransaction() method
- begin() method
- persist()
- close()

JPA EntityManager is supported by the following set of methods.

- **persist** – Make an instance managed and persistent.
- **merge** – Merge the state of the given entity into the current persistence context.
- **remove** – Remove the entity instance.
- **find** – Find by primary key. Search for an entity of the specified class and primary key. If the entity instance is contained in the persistence context, it is returned from there.

Collection Mapping

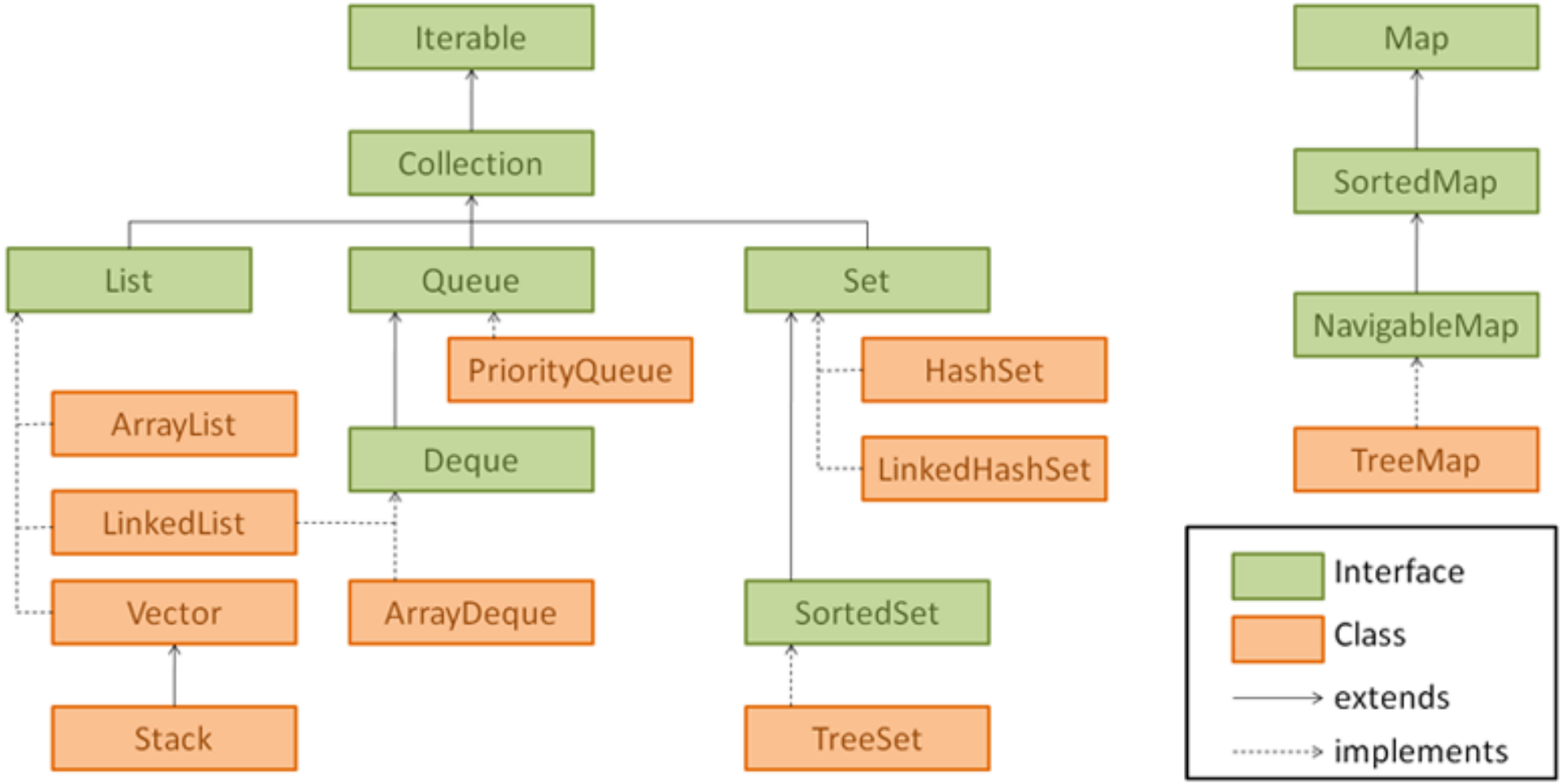
A Collection is a java framework that groups multiple objects into a single unit. It is used to store, retrieve and manipulate the aggregate data.

In JPA, we can persist the object of wrapper classes and String using collections.

JPA allows three kinds of objects to store in mapping collections - Basic Types, Entities and Embeddables.

Collection Types

- List
- Set
- Map



TRANSACTION

A transaction is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed.

These are called ACID properties -:

- Atomicity
- Consistency
- Isolation
- Durability

ACID Properties

Atomicity:
This property refers to the "all or nothing" nature of a transaction. A transaction must either complete successfully and commit all its changes to the database, or it must be rolled back and leave the database unchanged. In other words, a transaction must be treated as a single, indivisible unit of work.

Consistency:
This property ensures that a transaction brings the database from one valid state to another. In other words, the database must remain in a consistent state before and after the transaction is executed. This means that any constraints or rules defined in the database must be satisfied after the transaction is completed.

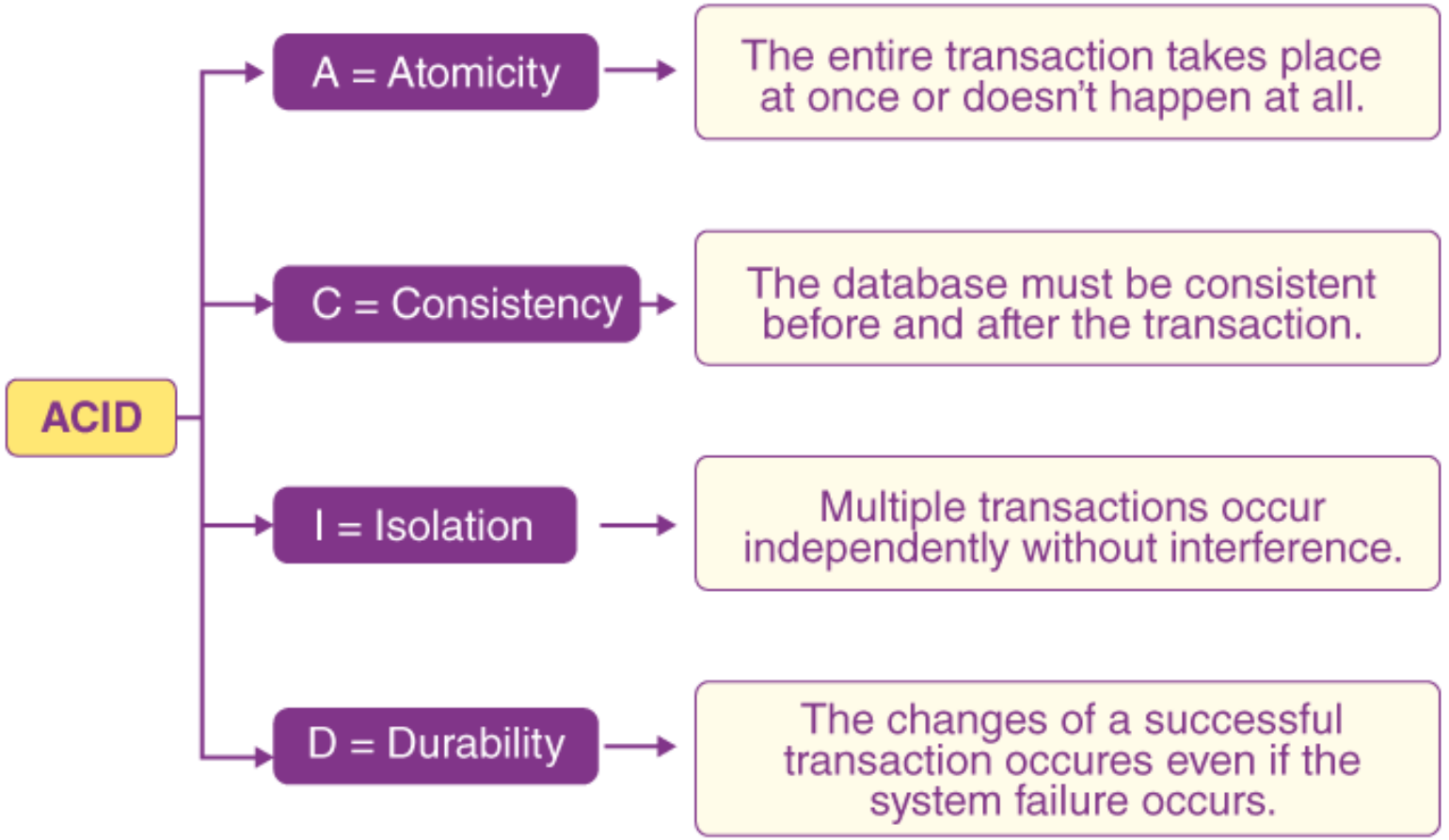
ACID Properties

Isolation:
This property ensures that concurrent transactions do not interfere with each other. Each transaction must be executed as if it were the only transaction in the system, even if there are multiple transactions running simultaneously. This means that the intermediate state of a transaction should not be visible to other transactions until it is committed.

Durability:
This property ensures that once a transaction is committed, its changes are permanent and will survive any subsequent system failures, such as power outages, crashes, or network failures. In other words, the database should be able to recover to the state it was in before the failure occurred, and all committed transactions should be persisted in the database.

The ACID properties are important because they ensure that database transactions are reliable and consistent. They are a fundamental aspect of database design and are essential for maintaining the integrity of data in a database system.

ACID Properties



Transaction Management Type in EJB

The Enterprise Bean Provider and the client application programmer are not exposed to the complexity of distributed transactions.

The Bean Provider can choose between using **programmatic transaction demarcation** in the enterprise bean code (this style is called **bean-managed transaction** demarcation) or **declarative transaction demarcation** performed automatically by the EJB container (this style is called **container-managed transaction** demarcation).

By default, a session bean or message-driven bean has container managed transaction demarcation if the transaction management type is not specified. The Bean Provider of a session bean or a message-driven bean can use the **TransactionManagement** annotation to declare transaction type . The value of the **TransactionManagement** annotation is either **CONTAINER** or **BEAN**.

Transaction Attributes

A transaction attribute controls the scope of a transaction.

A transaction attribute can have one of the following values:

- Required
- RequiresNew
- Mandatory
- NotSupported
- Supports
- Never

Required:
The method must run within a transaction. If an existing transaction exists when the method is called, it will be reused. Otherwise, a new transaction will be started.

RequiresNew:
The method must run in a new transaction. If an existing transaction exists when the method is called, it will be suspended until the method completes.

Mandatory:
The method must run within an existing transaction. If no transaction exists when the method is called, an exception will be thrown.

NotSupported:
The method must not run within a transaction. If an existing transaction exists when the method is called, it will be suspended until the method completes.

Supports:
The method can run with or without a transaction. If an existing transaction exists when the method is called, it will be reused. Otherwise, the method will run without a transaction.

Never:
The method must not run within a transaction. If an existing transaction exists when the method is called, an exception will be thrown.

Transaction Attribute	Client's Transaction	Business Method's Transaction
Required	None T1	T2 T1
RequiresNew	None T1	T2 T2
Mandatory	None T1	Error T1
NotSupported	None T1	None None
Supports	None T1	None T1
Never	None T1	None Error

TIMER SERVICE

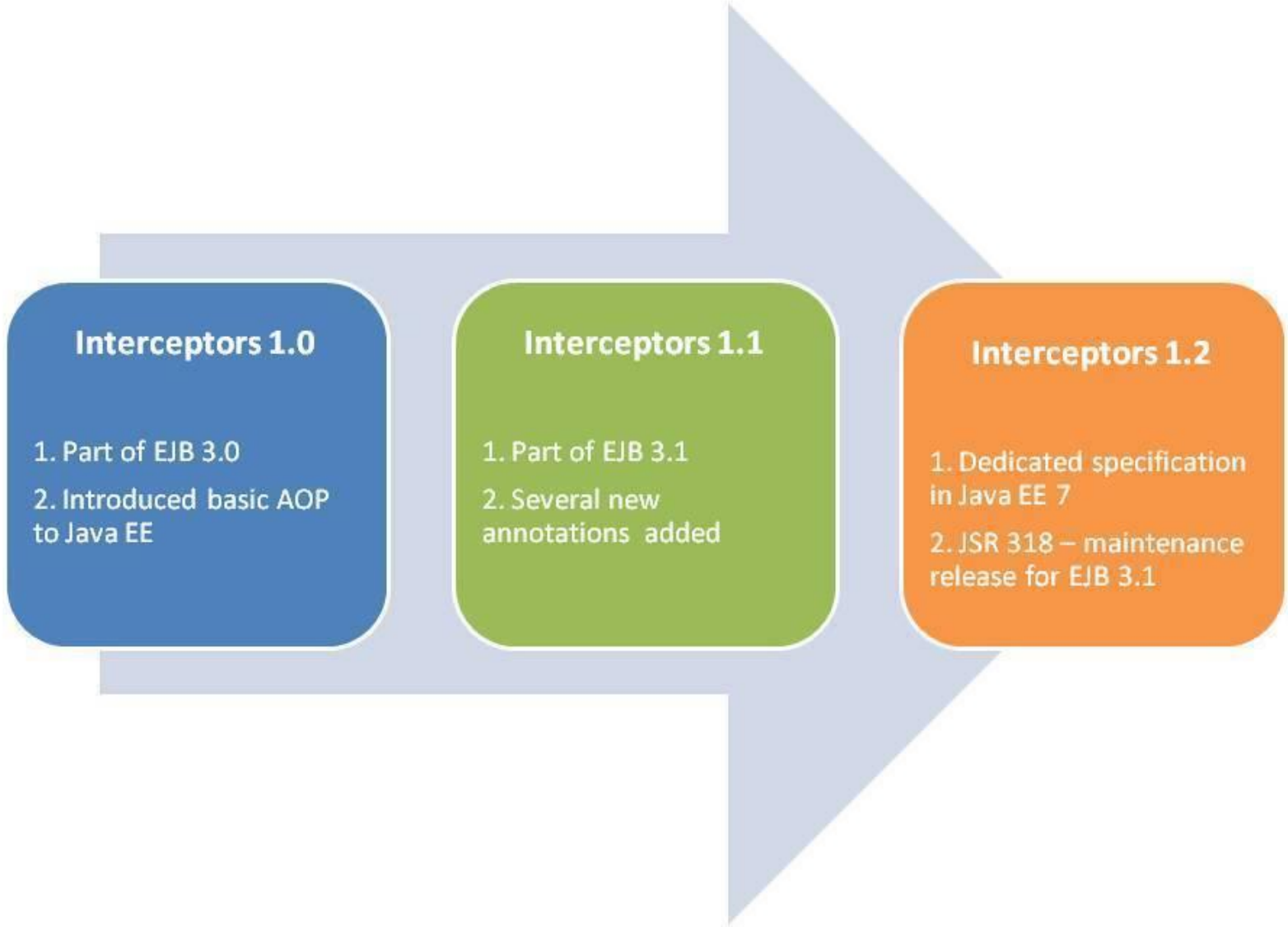
Applications that model business workflows often depend on timed notifications. The timer service of the enterprise bean container enables you to schedule timed notifications for all types of enterprise beans except for stateful session beans. You can schedule a timed notification to occur according to a calendar schedule, at a specific time, after a duration of time, or at timed intervals.

Enterprise bean timers are either programmatic timers or automatic timers. Programmatic timers are set by explicitly calling one of the timer creation methods of the **TimerService** interface. Automatic timers are created upon the successful deployment of an enterprise bean that contains a method annotated with the **java.ejb.Schedule** or **java.ejb.Schedules** annotations.

INTERCEPTORS

In J2EE (Jakarta EE), Interceptors are a powerful and elegant mechanism designed to handle cross-cutting concerns—aspects of a program that affect multiple components but are not part of the core business logic. Interceptors allow developers to write this logic in one place and apply it to many beans or methods, promoting code reuse, modularity, and separation of concerns.

Interceptors are used in conjunction with Java EE managed classes to allow developers to invoke interceptor methods on an associated target class, in conjunction with method invocations or lifecycle events. Common uses of interceptors are logging, auditing, and profiling.


Interceptors 1.0
1. Part of EJB 3.0
2. Introduced basic AOP to Java EE
Interceptors 1.1
1. Part of EJB 3.1
2. Several new annotations added
Interceptors 1.2
1. Dedicated specification in Java EE 7
2. JSR 318 – maintenance release for EJB 3.1

An **interceptor** is a special class or method that can "intercept" lifecycle events or business method calls in a component.

Purpose: To handle *cross-cutting concerns* like:

- Logging
- Security
- Auditing
- Performance Monitoring

Types of Interceptors

- Method Interceptors (@AroundInvoke)
- Lifecycle Interceptors (@PostConstruct, @PreDestroy)
- Constructor Interceptors (@AroundConstruct)

Interceptor Annotations Overview

Annotation	Description
@Interceptor	Marks the class as an interceptor
@AroundInvoke	Method called around business method
@AroundConstruct	Called during bean construction
@InterceptorBinding	CDI-style binding for reusability
@Priority	Defines execution order in CDI

SECURITY

Enterprise tier and web tier applications are made up of components that are deployed into various containers. These components are combined to build a multitier enterprise application. Security for components is provided by their containers.

Characteristics of Application Security

- Java EE applications consist of components that can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. Authorization provides controlled access to protected resources. Authorization is based on identification and authentication. Identification is a process that enables recognition of an entity by a system, and authentication is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

- Authorization and authentication are not required for an entity to access unprotected resources. Accessing a resource without authentication is referred to as unauthenticated, or anonymous, access.

The characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise include the following:

Authentication: The means by which communicating entities, such as client and server, prove to each other that they are acting on behalf of specific identities that are authorized for access. This ensures that users are who they say they are.

Authorization, or access control: The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. This ensures that users have permission to perform operations or access data.

Data integrity: The means used to prove that information has not been modified by a third party, an entity other than the source of the information. For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent. This ensures that only authorized users can modify data.

Confidentiality, or data privacy: The means used to ensure that information is made available only to users who are authorized to access it. This ensures that only authorized users can view sensitive data.

•**Non-repudiation**: The means used to prove that a user who performed some action cannot reasonably deny having done so. This ensures that transactions can be proved to have happened.

•**Quality of Service**: The means used to provide better service to selected network traffic over various technologies.

•**Auditing**: The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

JAX-RS

JAX-RS is Java API for RESTful Web Services (JAX-RS) is a Java programming language API spec that provides support in creating web services according to the Representational State Transfer (REST) architectural pattern.

There are two main implementations of JAX-RS API:

- Jersey Framework
- RESTEasy

JAX-RS Annotations Overview

Root Resource

Root resource classes are POJOs (Plain Old Java Objects) that are annotated with @Path have at least one method annotated with @Path or a resource method designator annotation such as @GET, @PUT, @POST, @DELETE.

Resource methods are methods of a resource class annotated with a resource method designator (resource method designator annotation such as @GET, @PUT, @POST, @DELETE).

HANDLING EXCEPTIONS

The exceptions thrown by enterprise beans fall into two categories:

- System Exceptions
- Application Exceptions

System Exceptions

A system exception indicates a problem with the services that support an application. For example, a connection to an external resource cannot be obtained, or an injected resource cannot be found. If it encounters a system-level problem, your enterprise bean should throw a **javax.ejb.EJBException**.

Because the **EJBException** is a subclass of the **RuntimeException**, you do not have to specify it in the throws clause of the method declaration. If a system exception is thrown, the EJB container might destroy the bean instance. Therefore, a system exception cannot be handled by the bean's client program, but instead requires intervention by a system administrator.

Application Exceptions

An application exception signals an error in the business logic of an enterprise bean. Application exceptions are typically exceptions that you've coded yourself, such as the **BookException** thrown by the business methods of the **CartBean** example. When an enterprise bean throws an application exception, the container does not wrap it in another exception. The client should be able to handle any application exception it receives.

If a system exception occurs within a transaction, the EJB container rolls back the transaction. However, if an application exception is thrown within a transaction, the container does not roll back the transaction.