Enterprise application development with Java EE

Java EE

- The Java technology
- Enterprise applications
- Java Persistence API
- Enterprise JavaBeans

The Java technology

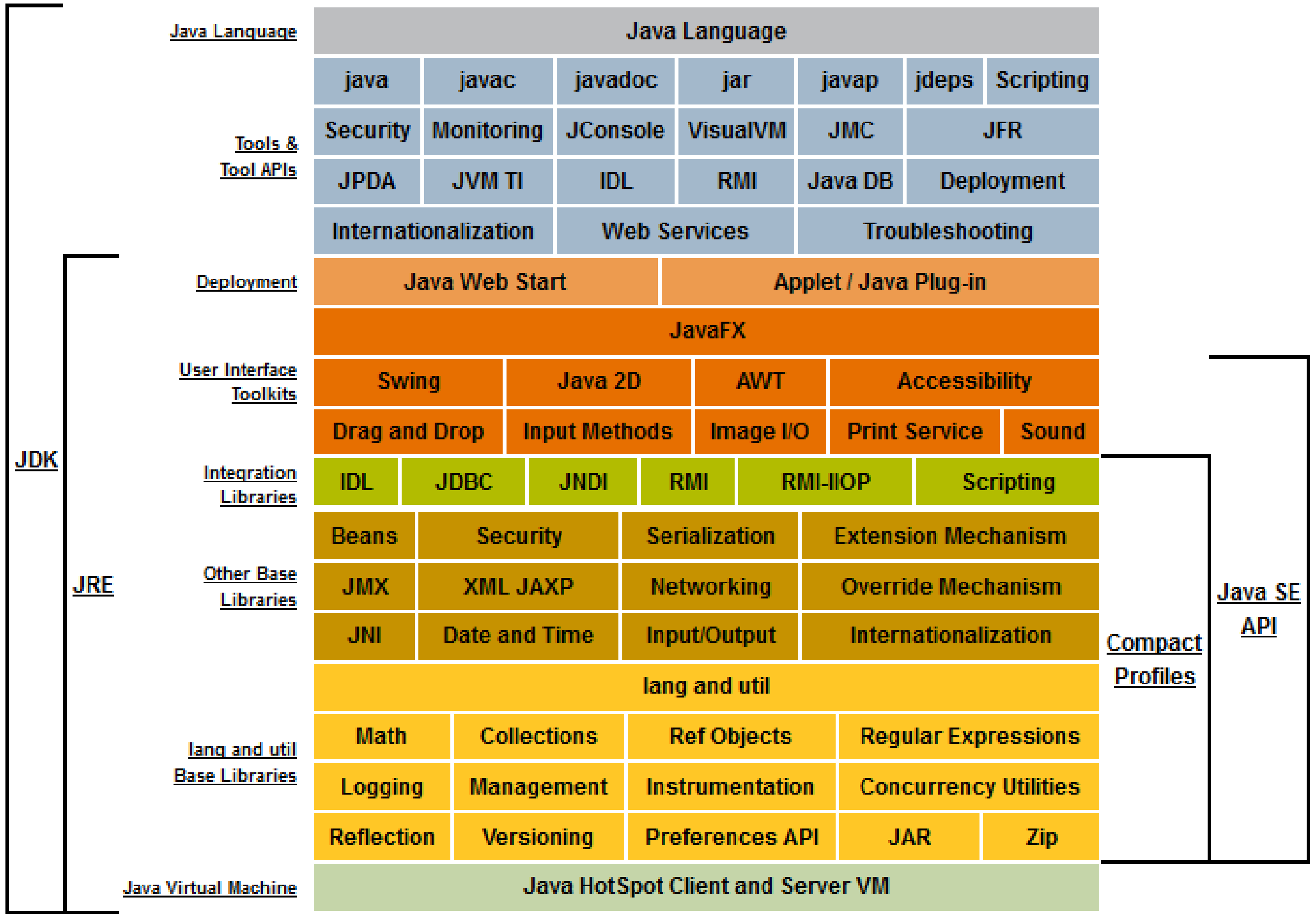
- Java technology includes a *programming language* and a *platform*
- *Java programming language* is a high-level object-oriented language, with a particular syntax and style
- *Java platform* is a particular environment in which Java applications run; it consists of:
  - *Java Virtual Machine* (JVM): an application written for a particular hardware and software platform that runs Java applications
  - *Application Programming Interface* (API): a collection of software components that are used to create applications and other software components

Java platform types

There are 3 Java platforms available:

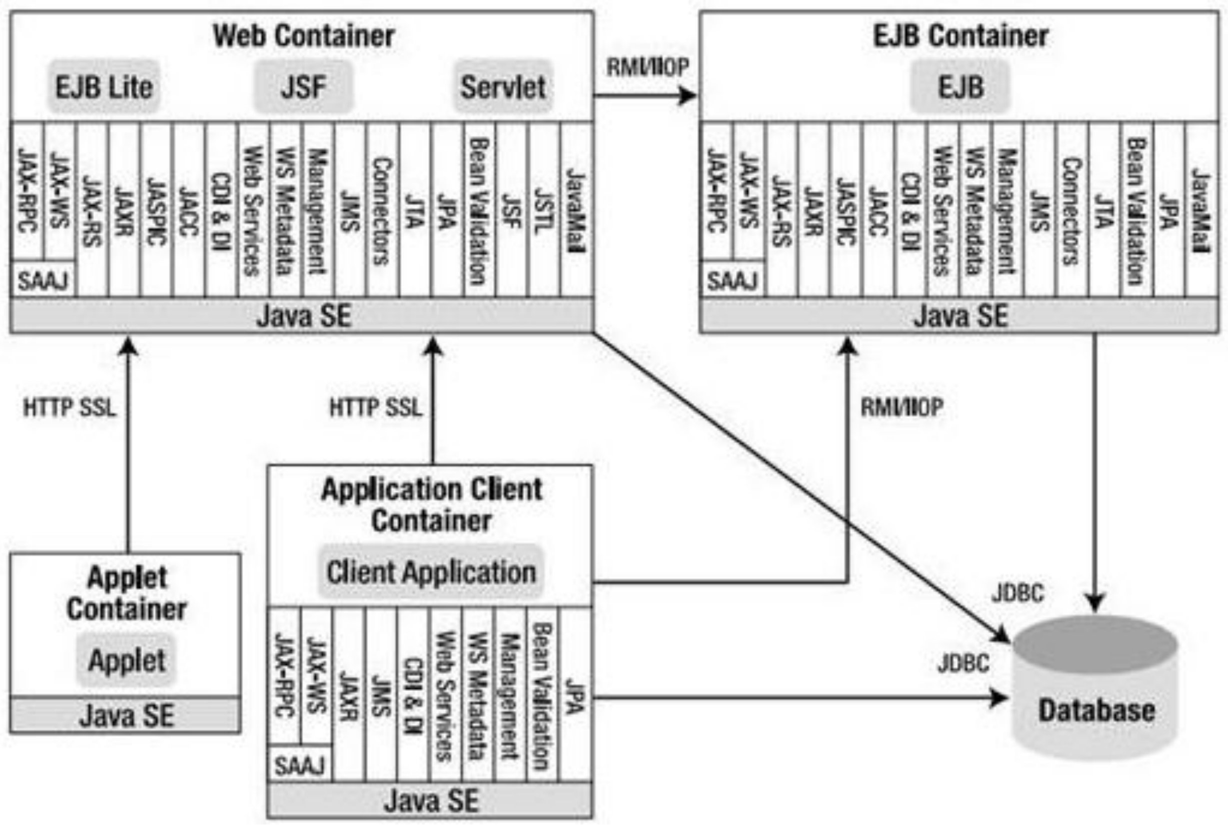
- *Java Platform, Standard Edition* (Java SE): provides the core functionality of the Java programming language, defining everything from the basic types to high-level classes
  - *Java FX*: a platform for creating GUIs
    - Used to be separate, merged with Java SE 8, unmerged again from Java SE 11 and open-sourced (openjfx.io)
- *Java Platform, Enterprise Edition* (Java EE): built on top of Java SE, Java EE is used for developing and running large-scale, multi-tiered network applications
  - Currently transitioning into Eclipse Enterprise for Java (EE4J)
- *Java Platform, Micro Edition* (Java ME): a small-footprint JVM for running Java applications on devices with limited resources, such as smart phones

The Java SE architecture



JDK
JRE
Java Language
Java Language
Tools & Tool APIs
java | javac | javadoc | jar | javap | jdeps | Scripting
Security | Monitoring | JConsole | VisualVM | JMC | JFR
JPDA | JVM TI | IDL | RMI | Java DB | Deployment
Internationalization | Web Services | Troubleshooting
Deployment
Java Web Start | Applet / Java Plug-in
User Interface Toolkits
JavaFX
Swing | Java 2D | AWT | Accessibility
Drag and Drop | Input Methods | Image I/O | Print Service | Sound
Integration Libraries
IDL | JDBC | JNDI | RMI | RMI-IIOP | Scripting
Other Base Libraries
Beans | Security | Serialization | Extension Mechanism
JMX | XML JAXP | Networking | Override Mechanism
JNI | Date and Time | Input/Output | Internationalization
lang and util Base Libraries
lang and util
Math | Collections | Ref Objects | Regular Expressions
Logging | Management | Instrumentation | Concurrency Utilities
Reflection | Versioning | Preferences API | JAR | Zip
Java Virtual Machine
Java HotSpot Client and Server VM
Compact Profiles
Java SE API

The Java EE architecture



Java EE

- The Java technology
- Enterprise applications
- Java Persistence API
- Enterprise JavaBeans

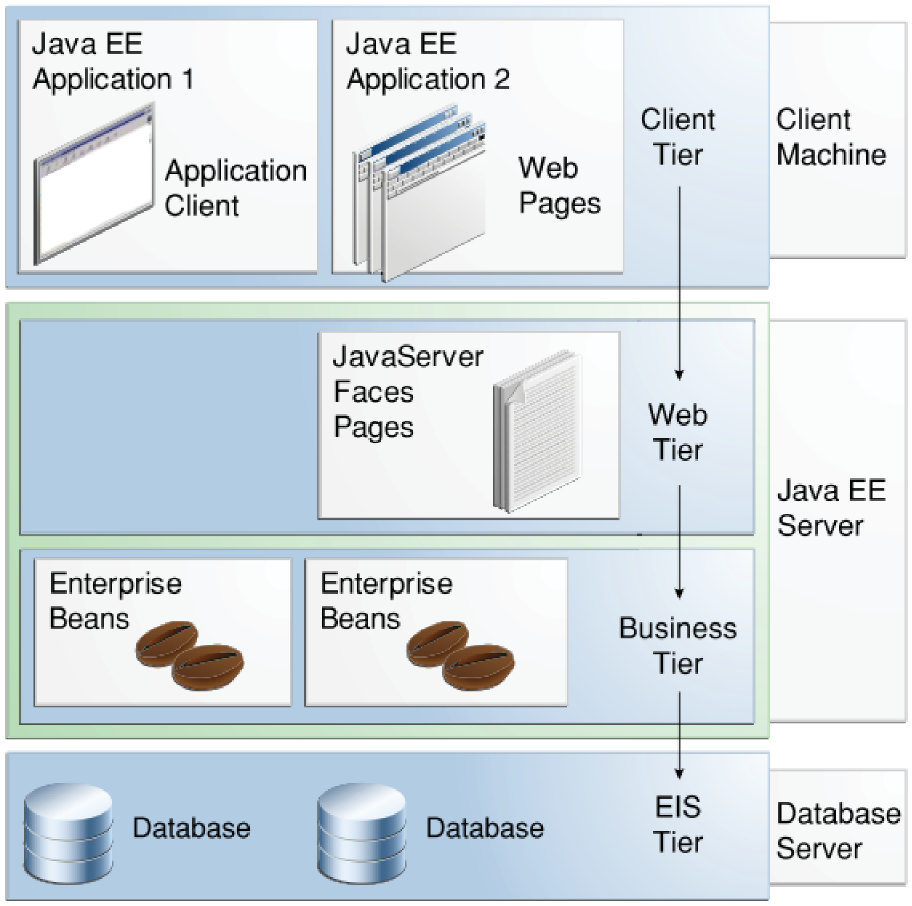
Enterprise applications

- The Java EE platform is designed to help developers create *large-scale*, *multi-tiered*, *scalable*, *reliable*, and *secure* networked applications
- These applications are often called *enterprise applications* because they are used to support the functioning of large enterprises
  - However, the benefits of enterprise applications can be helpful even for small organizations and individual developers
- The Java EE platform is designed to reduce the complexity of enterprise application development by providing a development model, API, and runtime environment

Multi-tiered applications

- The functionality of an enterprise application is divided into isolated functional areas called *tiers*
- A Java EE-based multi-tier application consists of the following 4 tiers: *client*, *web*, *business*, and *data* or *enterprise information system* (EIS) tier
  - However, Java EE-based applications are considered to be three-tiered, because they are distributed over three locations: *client machines*, the *Java EE server machine*, and the *database machine* at the back end
- Three-tiered applications extend the standard two-tiered client-server model by placing a multithreaded application server between the client application and back-end storage

Java EE multi-tiered architecture



Java EE server

- A *Java EE server* is an application that implements the Java EE platforms and provides the standard Java EE services
  - Java EE servers are often called *application servers*, because they allow developers to serve applications to clients, much like web servers offer web pages
- Traditionally, Java EE servers also provide support for transaction management, concurrency control, security authorization, and load balancing
- Currently there are 5 Java 8 EE certified servers: *(Eclipse) GlassFish, Wildfly, WebSphere, JBoss, InforSuite*
  - GlassFish 5 is the reference implementation for Java 8 EE

Tiers of Java EE-based applications

- The *client tier* runs on client machines in form of web browsers, applets, or standalone applications
  - Web-based clients communicate with the web tier, while standalone applications may access the business tier directly
- The *web tier* is an intermediate tier between web-based clients and the core application functionality
  - It hosts web applications made of *servlets*, *JavaServer Pages* (JSP), and *JavaServer Faces* (JSF)
- The *business tier* hosts the core functionality of enterprise applications
  - It employs *Enterprise Java Beans* (EJB) and *Web Services* (WS)
- The *EIS* tier consists of database servers, and is accessed from the Java EE server by using *Java Database Connectivity API* (JDBC), *Java Persistence API* (JPA), etc.

Java EE

- The Java technology
- Enterprise applications
- Java Persistence API
- Enterprise JavaBeans

Java Persistence API

- *Persistence* is a property of an object which allows it to save its runtime state and restore it the next time program starts
- Java offers two main ways for saving the state of an object:
  - Serialization
  - *Object-Relational Mapping* (ORM): allows developers to think in object-oriented terms of classes, interfaces, objects, etc. and delegate access to relational databases to external tools or frameworks
- *Java Persistence API* (JPA) is the standard Java EE ORM framework

Entities

- When talking about persistent objects, JPA specification uses the term *entity*
- In JPA, the term “object” signifies a POJO – a Plain Old Java Object
- An entity, on the other hand, is an object with additional properties:
  - It can be permanently stored in a database
  - It can be queried from the database
  - It follows a defined life-cycle

Object-relational mapping

- For its functioning, JPA relies on *metadata* associated with each entity
- This metadata is usually provided in form of annotations
  - For JPA, as well as for the whole Java EE platform, annotations represent one of the most important programming constructs
- The *javax.persistence* package defines a large number of annotations, some of which are:
  - *@Entity*: marks the annotated class as an entity
  - *@Id*: marks the entity field as the primary key; a primary key can be any Java primitive type, any primitive wrapper type, *String*, *Date*, *BigDecimal*, or *BigInteger*
  - *@GeneratedValue*: indicates that the field value should be auto-generated (e.g. for primary keys)

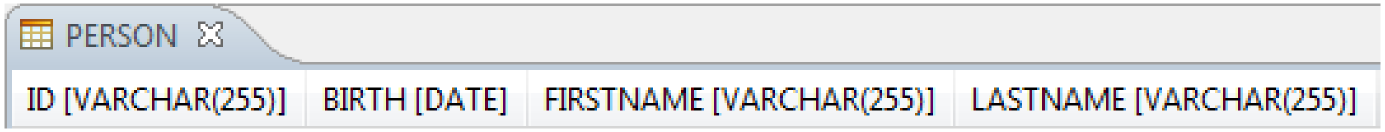
Entity example

- The following example shows an entity *Person*, with an auto-generated primary key *id*, first and last name, and a date of birth:

```
@Entity
public class Person {
   @Id
   @GeneratedValue
   private String id;
   private String firstName;
   private String lastName;
   // Date and Calendar types need to be annotated
   @Temporal(TemporalType.DATE)
   private Date birth;
}
```

Entity rules

- For each entity, a corresponding database table with the same name is created
- Each field of an entity becomes a column in the table, with JPA taking care of conversions between Java and SQL data types
  - *@Transient* annotation can be used to mark a field as transient (i.e. non-persistent)
- An entity must have a default (no-arguments) constructor
- For the *Person* entity, the following table is created in a *Derby* database:



PERSON

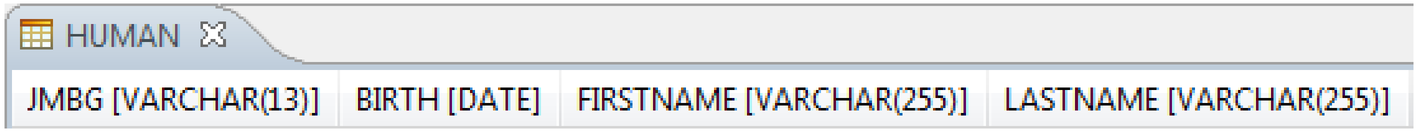
ID [VARCHAR(255)]	BIRTH [DATE]	FIRSTNAME [VARCHAR(255)]	LASTNAME [VARCHAR(255)]

Programming by exception

- JPA, like many other Java EE specifications, uses the concept of *programming by exception*
- The basic idea of this concept is that JPA follows certain default mapping rules
  - For example, the database table name matches the entity name, the table column name matches the entity field name, etc.
- Developers that need different behavior should provide additional metadata
  - Usually in form of new annotations or annotation arguments

Programming by exception – example

```
// table name should be Human instead of Person
@Entity(name = "Human")
public class Person {
  @Id
  @GeneratedValue
  // this column will now be called
  // JMBG and limited to 13 chars
  @Column(name = "JMBG", length = 13)
  private int id;
  private String firstName;
  private String lastName;
  @Temporal(TemporalType.DATE)
  private Date birth;
  // this is some non-persistent field
  @Transient
  private String desc;
}
```


HUMAN

JMBG [VARCHAR(13)]	BIRTH [DATE]	FIRSTNAME [VARCHAR(255)]	LASTNAME [VARCHAR(255)]

Persistence unit

- All entities of a JPA-based application should be listed in a *persistence unit*
- Additionally, a persistence unit contains a number of attributes that describe:
  - Which concrete implementation of JPA to use (e.g. *EclipseLink*, *TopLink*, *Hibernate*)
  - Name and driver of the target database, its physical location, username and password, etc.
  - How database transactions should managed
- The persistence unit is described in an external file named *persistence.xml* which usually doesn’t need to be edited manually

Persistence context

- A *persistence context* is a set of managed entity instances at a given time, for a given transaction
- It can be seen as a kind of a cache
  - When an application requests an entity, the persistence API will first check the application's current persistence context
  - If the entity is not found there, only then will the API try to load it from the database
- Entities live in the persistence context for the duration of the transaction
  - Once the transaction is committed, all changes in the persistence context are reflected in the database

Entity manager

- *Entity manager* is the object used to interact with the persistence context
- It manages states and life-cycles of entities, and it is used to query entities from a persistence context
- When an entity manager obtains a reference to an entity, it is said to be *managed*
  - Until that point, the entity is a *detached* POJO
- The strength of JPA is that entities can be used as regular objects by different layers of an application and become managed by the entity manager when you need to load or insert data into the database

The *EntityManager* interface

- The *EntityManager* interface represents the JPA entity manager; its methods include:
  - **void persist(Object entity)**: make an entity instance managed and persistent
  - **void detach(Object entity)**: remove the given entity from the persistence context, causing it to become detached
  - **void refresh(Object entity)**: refresh the state of the instance from the database, overwriting changes made to the entity, if any
  - **void remove(Object entity)**: remove the entity; once removed, the entity is deleted from the database, and detached from the entity manager
  - **<T> T find(Class<T> entityClass, Object primaryKey)**: searches for an entity of the specified class and primary key

Handling transactions

- Java SE-based applications that use JPA need to manage database transactions manually
- In Java EE-based applications, the Java EE server is in charge of managing transactions
- A transaction is created by invoking the *getTransaction()* method of an *EntityManager* instance
- Transactions are represented by the *EntityTransaction* interface, which includes the following methods:
  - **void begin()**: start a resource transaction
  - **void commit()**: commit the current resource transaction, writing any changes from the persistence context to the database
  - **void rollback()**: roll back the current resource transaction

Example – persisting a person

```
// create EntityManager for the given persistence unit
EntityManagerFactory emf = Persistence.
  createEntityManagerFactory("OOP2_2011-2012_EE");
EntityManager em = emf.createEntityManager();
try {
  // begin a transaction
  EntityTransaction et = em.getTransaction();
  et.begin();
  try {
    // create a person
    Person p = new Person();
    p.setFirstName("John");
    p.setLastName("Smith");
    p.setBirth(new Date(1969, 12, 21));
    // persist the person - it becomes managed
    em.persist(p);
    // commit this transaction - p will be saved to the DB
    et.commit();
  } catch (Exception ex) {
    ex.printStackTrace();
    et.rollback(); // error, rollback the transaction
  }
} finally {
  em.close(); // done
}
```

JPQL

- *Java Persistence Query Language* (JPQL) is used to define searches against persistent entities independent of the underlying database
- Its syntax is strongly based on the syntax of the *Structured Query Language* (SQL)
  - The main difference is that in SQL the results obtained are in the form of tables, whereas JPQL uses an entity or a collection of entities
- JPQL supports the following queries:
  - *SELECT*: for retrieving entities
  - *UPDATE*: for updating existing entities
  - *DELETE*: for deleting entities from the database

JPA queries

- The 3 main types of queries supported by JPA are:
  - *Dynamic*: the query string is build dynamically at runtime
  - *Named*: specified at compile time
  - *Native*: used to execute a native SQL statement instead of a JPQL statement
- Named queries are more efficient than dynamic queries
  - This is because the persistence provider can translate the JPQL string to SQL once the application starts, rather than every time the query is executed
- Native queries should be used only if the application targets a specific database

Managing queries

- Regardless of the type, JPA queries are represented by the *Query* interface
- The interface offers a rich set of methods for query manipulation:
  - For executing a *SELECT* query, applications can use methods *getResultList()* and *getSingleResult()*
  - For executing an *UPDATE* or *DELETE* query, applications can invoke the *executeUpdate()* method; the method returns the number of entities updated or deleted
- In addition, JPA offers the *TypedQuery*, which is a generic version of *Query*

Creating a query

- *Query* instances are obtained from an *EntityManager* instance:
  - **Query createQuery(String jpqlQuery)**: create a dynamic query, based on the given JQPL string
  - **Query createNativeQuery(String sqlQuery)**: create a native query, based on the given SQL string
  - **Query createNamedQuery(String name)**: create a named query from the given name
- Each method has a generic counterpart, which returns an instance of the *TypedQuery* type

Example – using queries

- Named queries are defined inside a *@NamedQuery* annotation, which takes two arguments: the name of the query, and its content
- Multiple named queries for an entity should be specified inside of a *@NamedQueries* annotation
- The *Person* entity is updated to include two named queries – one that returns all persons, and one that returns persons whose name starts with a prefix:

```
@Entity
@NamedQueries(value = {
  @NamedQuery(name = "getAll", query = "SELECT p FROM Person p"),
  @NamedQuery(name = "getByPrefix",
    query = "SELECT p FROM Person p WHERE p.firstName LIKE :arg")
})
public class Person {
  ...
```

Example – using queries (cont')

```java
// create EntityManager for the given persistence unit
EntityManagerFactory emf = Persistence.
  createEntityManagerFactory("OOP2_2011-2012_EE");
EntityManager em = emf.createEntityManager();
try {
  // find all persons
  TypedQuery<Person> q = em.
    createNamedQuery("getAll", Person.class);
  List<Person> list = q.getResultList();
  for (Person p : list)
    System.out.println(p);

  // find persons whose name starts with "Jo"
  q = em.createNamedQuery("getByPrefix", Person.class);
  // set parameter "arg" of this named query
  q.setParameter("arg", "Jo%");
  list = q.getResultList();
  for (Person p : list)
    System.out.println(p);
} finally {
  em.close();
}
```

Java EE

- The Java technology
- Enterprise applications
- Java Persistence API
- Enterprise JavaBeans

Enterprise JavaBeans

- *Enterprise JavaBeans* (EJB) are managed, server-side components that encapsulate business logic of enterprise applications
  - They are used to build business layers of multi-tiered applications
- The EJB technology was introduced to handle common developing concerns as such as persistence, transactional integrity, and security in a standard way
  - This enables programmers to concentrate on the particular problem at hand

EJB containers

- EJBs execute inside a *container* – a runtime environment that provides common features to many enterprise applications
- Features that should be offered by an EJB container include:
  - *Remote client communication*: without writing any complex code, an EJB client can invoke methods remotely via standard protocols
  - *Dependency injection*: the container can inject several resources into an EJB
  - *State management*: the container can manage states of some EJBs for a particular client transparently
  - *Pooling*: the container creates a pool of instances that can be shared by multiple clients; once invoked, an EJB returns to the pool to be reused instead of being destroyed
  - *Component life-cycle*

EJB containers (cont')

- *Asynchronous calls*: some EJBs can communicate with other components (and vice versa) using asynchronous method invocations or messaging
- *Transaction management*: the container takes care of the commit or the rollback
- *Security*: class or method-level access control can be specified on EJBs to enforce user and role authentication
- *Concurrency support*: most of EJBs are thread-safe by nature

- Containers run inside application servers, which, additionally, introduce the support for scalability, load balancing, transparent failover, administration, monitoring, caching, etc.

Session and message-driven beans

- To help combat the complexity of enterprise applications, the Java EE platform defines two main types of EJBs: *session beans* and *message-driven beans* (MDB)
- Session beans represent the most important part of EJB technology: they encapsulate high-level business logic
- Message-driven beans are business objects whose execution is triggered by messages instead of by method calls
  - They were added to support event-driven programming
- Usually, MDBs just listen for incoming messages and invoke session beans accordingly

Types of session beans

- Based on the required functionality, an application can choose from the following types of session beans:
  - *Stateless*: does not maintain any conversational state with a client application; it is used to handle tasks that can be concluded with a single method call
  - *Stateful*: maintains state and is associated with a specific client; it is useful for tasks that have to be done in several steps
  - *Singleton*: there is only one instance of a singleton session bean for the entire application

Stateless beans

- Stateless beans are the most popular session bean components
- They are ideal for implementing a task that can be concluded with a single method call
- Stateless beans can be *pooled* and shared by several clients:
  - For each stateless EJB, the container keeps a certain number of instances in memory (i.e. a *pool*)
  - When a client invokes a method on a stateless bean, the container picks up an instance from the pool and assigns it to the client
  - When the client request finishes, the instance returns to the pool to be reused

Stateless beans efficiency

- Stateless beans are efficient because only a small number of beans are needed to handle many clients
- The same instance can be recycled over and over, reducing the amount of required memory and the number of memory (de)allocations
- However, the client is unaware of the actual instance it interacts with
  - Even consecutive method calls by the same client can end up in different stateless bean instances!

Stateless bean example

- In order to define a stateless bean, it is enough to use *@Stateless* annotation:

```
import javax.ejb.Stateless;

@Stateless
public class StatelessBean {
  private String id;
  private static int counter;

  public StatelessBean() {
    ++counter;
    id = "Bean_" + counter;
    System.out.println("Stateless bean " + id + " created.");
  }

  public String getId() { return id; }
}
```

- Example output from creating 100 threads that invoke this bean:

```
Stateless bean Bean_1 created.
Stateless bean Bean_2 created.
Stateless bean Bean_3 created.
```

Stateful beans

- Stateful session beans preserve conversational state with the client: they are useful for tasks that have to be done in several steps
- When a client invokes a stateful bean on the server, the EJB container needs to provide the same instance for each subsequent method invocation
- Stateful beans cannot be reused by other clients, which means that there need to be as many stateful bean instances as there are clients
- To avoid a large memory footprint, servers use the *passivation and activation* technique
  - Passivation is the process of removing an instance from memory and saving it to a persistent location, before the next request from the client
  - Activation is the inverse process of restoring the state and applying it to an instance

Annotations for stateful beans

- A stateful bean is declared by annotating a POJO with *@Stateful*
- The container automatically removes a stateful bean instance when the client's session ends or expires
- However, developers can control when an instance should be removed, by using two additional annotations:
  - *@Remove*: annotates a method; causes the bean instance to be permanently removed from memory after the method is invoked
  - *@StatefulTimeout*: assigns a timeout value, which is the duration the bean is permitted to remain idle before being removed by the container

Stateful bean example

- The following example simulates a shopping cart
  - It enables the client to include and exclude items
  - Once finished with the shopping, the client invokes the *checkout()* command

```java
@Stateful
public class ShoppingCartEJB {
  // list of items in the cart
  private List<Object> cartItems = new ArrayList<Object>();

  public void addItem(Object item) { cartItems.add(item); }

  public void removeItem(Object item) { cartItems.remove(item); }

  @Remove
  public void checkout() {
    // perform checkout
    // the bean is removed once this method completes
  }
}
```

Singleton beans

- A singleton bean is a session bean that is instantiated once per application
  - In a shopping cart scenario, a singleton bean may be used to represent a global item price list
  - This would prevent the application from having to perform database queries for each new client
- In order to define a singleton session bean, it is enough to annotate a POJO with *@Singleton*
  - The EJB container ensures that only one instance is created

Singleton initialization

- By default, a singleton bean is created the first time a client uses it
- However, the initialization can be forced to occur during the application’s startup
- This behavior is achieved by annotating the bean class with *@Startup*
- If multiple singleton beans are used to initialize data for an application, and must be initialized in a specific order, the *@DependsOn* annotation is used
  - The annotation accepts one or more strings indicating names of singleton beans that must be initialized before the annotated one

Singleton concurrency

- Unlike other types of session beans, singletons can and are accessed concurrently
- There are two main ways of controlling the concurrent access:
  - *Container-managed concurrency* (CMC): the container controls concurrent access to the bean instance based on metadata
  - *Bean-managed concurrency* (BMC): the container allows full concurrent access and defers the synchronization responsibility to the bean
- CMC is the default and the preferred behavior

Container-managed concurrency

- With CMC, developers use the *@Lock* annotation to specify how the container must manage concurrency when a client invokes a method
- The annotation can take the following values:
  - *@Lock(LockType.WRITE)*: exclusive write lock; this is the default locking attribute
  - *@Lock(LockType.READ)*: shared read lock
- @Lock can be specified on the class, or the methods
  - If specified on the class, it applies to all methods
- To avoid blocking the client indefinitely, applications can use the *@AccessTimeout* annotation
  - The annotation indicates rejecting a request if the lock is not acquired within a certain time interval

Singleton bean example

- The following bean simulates a global item price list for an online shopping application

```
@Singleton
@Startup // run when the application starts
public class PriceListEJB {
  // a map of items and their prices
  private Map<Object, Double> prices;

  public PriceListEJB() {
    // load prices from the database
  }

  @Lock(LockType.READ)
  @AccessTimeout(value=30, unit=TimeUnit.SECONDS) // give up after 30s
  public Double getPrice(Object item) {
    return prices.get(item);
  }

  @Lock(LockType.WRITE)
  public void updatePrice(Object item, Double newPrice) {
    prices.put(item, newPrice); // then save to db
  }
}
```

Remote and local access

- Session beans can be designed for local use (i.e. from the same JVM), for remote use, or both
- Remote functionality of a session bean must be specified in an interface annotated with *@Remote*
- Local functionality can be specified in an interface annotated with *@Local*
  - Alternatively, the bean itself can be annotated with *@LocalBean*
- Remote and local session bean interfaces are often called *business interfaces*, as they contain the declaration of business methods that are visible to the client and implemented by the bean class

Dependency injection

- *Dependency injection* (DI) is a mechanism used to *inject* references of resources into fields
- The basic idea of DI is that instead of hard-coding the dependencies, a component just lists all the services it requires, and the DI framework supplies them
  - Dependencies include references to other components, EJBs, appropriate database driver, etc.
- Java EE uses annotations for DI, some of which are:
  - *@EJB*: injects a reference of an EJB into the annotated variable
  - *@PersistenceContext*: injects an entity manager instance into the annotated variable

Shopping cart example

- The following example demonstrates an implementation of a web-based shopping cart
- It includes the following set of components:
  - Entity *User*: registered users of the system
  - Entity *Item*: items on sale
  - Singleton bean *ItemList*
  - Stateful bean *ShoppingCart*

Shopping cart – entities

```
// user is a reserved word in SQL
@Entity(name = "CartUser")
public class User {
  @Id
  private String name;
  private String password;
  // list of purchased items
  private List<Item> items;
}

@Entity
@NamedQuery(name = "Item.getAll",
            query = "SELECT i FROM Item i")
public class Item implements Serializable {
  private static final long serialVersionUID = 1L;
  @Id  @GeneratedValue
  private int id;
  private String description;
  private double price;
}
```

Shopping cart – *ItemList* singleton bean

```
@Singleton
@LocalBean // for local use only
@Lock(LockType.READ)
public class ItemList {
  // dependency injection: an EntityManager instance
  // will be automatically injected into "em" when needed
  @PersistenceContext(name = "OOP2_2011-2012_EE")
  private EntityManager em;
  // list of available items
  private List<Item> items;

  public List<Item> get() {
    if (items == null) { // lazy loading
      TypedQuery<Item> q = em
        .createNamedQuery("Item.getAll", Item.class);
      items = q.getResultList();
    }
    return items;
  }
}
```

Shopping cart – the business interface

```
@Remote
public interface ShoppingCartI extends Serializable {

  public boolean login(String name, String password);

  public List<Item> getItemList();

  public double addToCart(Item i); // returns the total price

  public void removeFromCart(Item i);

  public void checkout();
}
```

The *ShoppingCart* bean

```
@Stateful
public class ShoppingCart implements ShoppingCartI {
  private static final long serialVersionUID = 1L;
  // dependency injection
  @PersistenceContext(name = "OOP2_2011-2012_EE")
  private EntityManager em;
  // dependency injection: an ItemList instance will be
  // automatically injected into "priceList" when needed
  @EJB
  private ItemList priceList;
  private List<Item> cart; // selected items
  private double totalPrice;
  private User activeUser;

  @Override
  public boolean login(String name, String password) {
    if (activeUser != null) return false; // already logged in
    User u = em.find(User.class, name);
    if ((u == null) || !u.getPassword().equals(password))
      return false; // no such user, or bad password
    activeUser = u;
    cart = new ArrayList<Item>();
    return true;
  }
```

The *ShoppingCart* bean (cont’)

```
  @Override public List<Item> getItemList() {
    if (activeUser == null) return null;
    return priceList.get();
  }

  @Override public double addToCart(Item i) {
    if (activeUser == null) return -1.0;
    cart.add(i);
    return totalPrice += i.getPrice();
  }

  @Override public void removeFromCart(Item i) {
    if (cart.remove(i))
      totalPrice -= i.getPrice();
  }

  @Remove // destroy the bean after executing this method
  @Override public void checkout() {
    activeUser.getItems().addAll(cart);
    em.merge(activeUser); // save changes to DB
  }
}
```