Overview

- interface for accessing relational databases
- unified
  - database independent
    - database vendor must provide a JDBC driver
- allows
  - executing SQL queries
  - accessing results of queries
    - similar to the reflection API
- packages
  - java.sql, javax.sql
- versions
  - JDK 1.4 – JDBC 3.0
  - JDK 6  – JDBC 4.0
  - JDK 7  – JDBC 4.1
  - JDK 7  – JDBC 4.2
  - JDK 9, 10  – JDBC 4.3
JDBC Driver

• JDBC API
  – in fact only interfaces
  – an implementation is provided via the driver
• driver
  – explicitly loaded and registered
  – Class.forName("com.driver.Name");
• after the driver is loaded, a connection to DB is created
  – Connection con = DriverManager.getConnection(url, 
    "myLogin", "myPassword");
  – url
    • jdbc:mysql://localhost/test
    • jdbc:odbc:source
JDBC Driver

- types of drivers
  - Native-protocol pure Java driver
  - Native-API partly-Java driver
  - JDBC-Net pure Java driver
  - JDBC-ODBC bridge plus ODBC driver
Basic classes and interfaces

- DriverManager – class
  - all methods are static
  - getConnection()
    - several variants
  - getDrivers()
    - all loaded drivers
  - getLogWriter(), setLogWriter()
  - println()
    - printing to a log
  - getLoginTimeout(), setLoginTimeout()
Basic classes and interfaces

- Connection – interface
  - creating and executing queries
- ResultSet – interface
  - query results
Class.forName("com.mysql.jdbc.Driver");
Connection con = DriverManager.getConnection("jdbc:mysql://localhost/test", "" , "" );

Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT * FROM test");

while (rs.next()) {
    // processing results line-by-line
}

stmt.close();
con.close();
Accessing results

- similar to the reflection API
  - `getString()`, `getInt()`, ...
  - work with current line
  - identification of a column by
    - name
    - order

```java
ResultSet rs = stmt.executeQuery("SELECT ID, NAME FROM TEST");

while (rs.next()) {
    int id = rs.getInt("ID");
    String s = rs.getString("STRING");
    System.out.println(id + " " + s);
}
```
Accessing results

- ResultSet.next()  
  - must be called even for the first line
- getString()  
  - can be called to all types  
    - with exception of SQL3 types  
  - automatic conversion to String
Queries

- `Connection.createStatement()`
  - ("empty") query creation

- `Statement.executeQuery("....")`
  - for queries returning results (SELECT)
  - results via ResultSet

- `Statement.executeUpdate("...")`
  - for queries returning no results
    - UPDATE
    - CREATE TABLE
    - ...
PreparedStatement

• PreparedStatement
  – interface
  – extends Statement
  – a prepared query with parameters
    • set before execution
  – methods
    • setType(int index, type v)
    • clearParameters()

```java
PreparedStatement pstmt =
    con.prepareStatement("UPDATE EMPLOYEES SET
    SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00)
pstmt.setInt(2, 110592)
```
Transactions

- by default – auto-commit mode
  - `commit` is performed after each change
- auto-commit can be unset

```java
con.setAutoCommit(false); //
// a sequence of queries
//
con.commit();    // or  con.rollback()
con.setAutoCommit(true);
```
Callable Statements

- access to stored procedures
- extends PreparedStatement
  - setting parameters
    - `setType(int index, type v)`
  - returning type must be registered
    - `registerOutParameter(int index, int sqlType)`
  - format
    a) `{?= call <procedure-name>[<arg1>,<arg2>, …]}`
    b) `{call <procedure-name>[<arg1>,<arg2>, …]}`

```java
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```
Handling errors

- SQLException
  - and its children
  - String getSQLState()
    - defined by X/Open
  - int getErrorCode()
    - specific for particular database

- warnings
  - SQLWarning
  - it is not an exception
  - must be explicitly tested
    - Statement.getWarnings()
    - SQLWarning.getNextWarning()
Batch update

- handling several queries together
- `Statement.addBatch(String sql)`
  - adds a query to the batch
- `int[] Statement.executeBatch();`
  - executes the batch
  - returns a number of affected lines for each query in the batch
Updatable ResultSet

- the default ResultSet cannot be changed, can be iterated only forward
  - can be changed when the Statement is created

```java
Statement stmt = con.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("SELECT ...");
```

- the resulting ResultSet can be changed, iterated freely
  - changes from different users are not visible in it
Java DB

- [Link](http://www.oracle.com/technetwork/java/javadb/)
- distributed with Java 7 a 8
  - not included in Java 9+

- Oracle supported distribution of the **Apache Derby** database
  - [Link](http://db.apache.org/derby/)
- fully in Java
- can be used
  - as a standalone server
  - embedded in an application
- targets small usage of memory (~ 2MB)

- there are also other databases in Java
Object databases

- non-relational databases
- storing and querying objects
- own access without JDBC

- NeoDatis
- db4o
- …

- an example for NeoDatis

```java
Sport sport = new Sport("volley-ball");
ODB odb = ODBFactory.open("test.neodatis");
odb.store(sport);
Objects<Player> players = odb.getObjects(Player.class);
odb.close();
```

- not much used
ORM

• an issue with OO databases
  - easy usage
  - lower performance, smaller support
• solution – ORM
  - object-relational mapping
  - a layer mapping a relational database to objects
  - roughly
    • class ~ a table scheme
    • object ~ row in a table
  - JDBC is typically used internally
    • automatically

• Hibernate
  • http://hibernate.org/
  • the most used ORM for Java
    • also implementations for different platforms
Document-oriented databases

- storing documents
  - semi-structured date

- MongoDB
  - https://www.mongodb.com/
  - documents ~ JSON

```java
MongoClient mongoClient = new MongoClient();
MongoDatabase database = mongoClient.getDatabase("mydb");
MongoCollection<Document> collection =
database.getCollection("test");
Document doc = new Document("name", "MongoDB")
    .append("type", "database")
    .append("count", 1)
    .append("versions", Arrays.asList("v3.2", "v3.0", "v2.6"))
    .append("info", new Document("x", 203).append("y", 102));
collection.insertOne(doc);
```
Java, summer semester 2018

JAVA

Hibernate
Architecture

Data Access Layer

Java Persistence API  Hibernate Native API

Hibernate

JDBC

Relational Database

image source: http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html
Core API

• Session
  – interconnection between DB and application
  – keeps inside a connection to DB
    • a JDBC connection
  – manages objects
    • contains a cache of objects

• SessionFactory
  – a session creator
  – contains mapping between objects and DB
  – can contain a cache of objects

• persistent objects
  – POJOs
  – should follow JavaBeans rules
    • but it is not necessary
Usage

● roughly
  - creating a configuration
    • XML
  - creating classes
    • Java
  - creating a mapping
    • XML, or
    • Java annotations
Configuration

- an XML file
- defines
  - a DB connection
  - a type of DB (dialect)
  - a mapping reference
  -...

```xml
<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">org.h2.Driver</property>
    <property name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"/>

    <property name="connection.pool_size">1</property>

    <property name="dialect">org.hibernate.dialect.H2Dialect</property>

    <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <property name="show_sql">true</property>

    <property name="hbm2ddl.auto">create</property>

    <mapping resource="org/hibernate/tutorial/hbm/Event.hbm.xml"/>
  </session-factory>
</hibernate-configuration>
```
Classes for persistent data

- POJO
- should follow JavaBeans rules for naming
  - it is not necessary
- a constructor without parameters is necessary
  - its visibility is not important

```java
class Event {
    private Long id;
    private String title;
    private Date date;

    public Event() {}
    public Event(String title, Date date) {
        this.title = title;
        this.date = date;
    }

    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```
Mapping

- an XML file
- mapping class attributes and columns
- defines
  - name
  - type
    - not necessary if it is “obvious”
    - Hibernate types
      - nor Java nor SQL types
      - they are “converters” between Java and SQL types
  - column
    - not necessary if it is the same as the name

```xml
<hibernate-mapping package="org.hibernate.tutorial.hbm">
  <class name="Event" table="EVENTS">
    <id name="id" column="EVENT_ID">
      <generator class="increment"/>
    </id>
    <property name="date" type="timestamp" column="EVENT_DATE"/>
    <property name="title"/>
  </class>
</hibernate-mapping>
```
Mapping

@javax.persistence.Entity
@javax.persistence.Table(name = "EVENTS")
public class Event {
  private Long id;
  private String title;
  private Date date;

  public Event() {
  }

  public Event(String title, Date date) {
    this.title = title;
    this.date = date;
  }

  @javax.persistence.Id
  @javax.persistence.GeneratedValue(generator="increment")
  @javax.persistence.GenerationType("increment")
  public Long getId() {  return id;  }

  private void setId(Long id) {  this.id = id;  }

  @javax.persistence.Temporal(javax.persistence.TemporalType.TIMESTAMP)
  @javax.persistence.Column(name = "EVENT_DATE")
  public Date getDate() { return date; }

  public void setDate(Date date) {  this.date = date;  }

  public String getTitle() { return title; }
  public void setTitle(String title) { this.title = title; }
}
Usage

- SessionFactory sessionFactory =
  new Configuration().configure().buildSessionFactory();

- Session session = sessionFactory.openSession();
  session.beginTransaction();
  session.save(new Event("Our very first event!", new Date()));
  session.save(new Event("A follow up event", new Date()));
  session.getTransaction().commit();
  session.close();

- List result = session.createQuery("from Event").list();
States of objects

- Transient
  - created object (new)
  - not yet associated with a Hibernate session

- Persistent
  - the object is associated with a session
    - created and then saved or loaded

- Detached
  - a persistent object but its session was terminated
  - can be associated with a new session
Using objects

- loading
  - `sess.load(Event.class, new Long(id));`
    - an exception is thrown if the object does not exist
    - may not immediately access DB
  - `sess.get(Event.class, new Long(id));`
    - returns null if the object does not exist

- querying
  - `sess.createQuery(...).list()`

- changing objects
  - `Event e = sess.load(Event.class, new Long(69));`
    - `e.set...`
    - `sess.flush();`
Using objects

• modifying detached objects
  – Event e = sess.load( Event.class, new Long(69) );
    e.set...
    ...
    secondSess.update(e);

• deleting objects
  – sess.delete(e);
Querying

- HQL – Hibernate query language
  - similar to SQL

```java
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

- native SQL can be used too

```java
sess.createSQLQuery("SELECT * FROM CATS").list();
```
Hibernate...

- other parts
  - creating classes from tables
  - support for full-text searching
  - object versioning
  - object validation
  - support of JPA (Java Persistence API)
  - ...
  - ...
XML processing
Overview

- JAXP – Java API for XML Processing
  - reading, writing and transforming XML
  - SAX, DOM, XSLT
    - according to W3C
    - supports multiple implementations
      - a reference implementation is a part of JDK
        - another one can be used

- JDOM
  - http://www.jdom.org/
  - „simplified“ DOM for Java

- JAXB – Java Architecture for XML Binding
  - mapping XML <=> Java objects

- Elliotte Rusty Harold: Processing XML with Java
  - http://www.cafeconleche.org/books/xmljava/
  - a freely accessible book
JAXP – overview

• packages
  - javax.xml.parsers
  - org.w3c.dom
  - org.xml.sax
  - javax.xml.transform

• SAX (Simple API for XML)
  - a “walk” through an XML document – element by element
  - each element can be processed
  - fast, low memory consumption
  - more complex to be used

• DOM
  - creates a tree in a memory from the document
  - easy to be used
  - slow, bigger memory consumption
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();

// vytvoří celý strom v paměti
Document document = builder.parse("file.xml");

Element root = document.getDocumentElement();
NodeList nl = root.getChildNodes();
for (int i=0; i<nl.length(); i++) {
    Node n = nl.item(i);
    ...
}
SAX: usage

class MyHandler extends DefaultHandler {
    void startDocument() {
        ...
    }
    void endDocument() {
        ...
    }
    void startElement(...)
    }
...

SAXParserFactory factory =
    SAXParserFactory.newInstance();
SAXParser saxParser = factory.newSAXParser();
saxParser.parse("file.xml", new MyHandler());
Implementation

- different implementations of JAXP exist
- `DocumentBuilderFactory.newInstance()` and `SAXParserFactory.newInstance()`
  - internally use the `ServiceLoader`
  - a variant
    `newInstance(String factoryClassName, ClassLoader classLoader)`
    - looks for a given class
JDOM – Overview

- http://www.jdom.org/
- API for XML
- directly for Java
  - uses std. API of Java (collections,...)
- easy to be used
- fast
- light-weight
Usage

SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(filename);
Element root = doc.getRootElement();

List children = current.getChildren();
Iterator iterator = children.iterator();
while (iterator.hasNext()) {
    Element child = (Element) iterator.next();
    ...
}

JAXB – overview

- automated mapping between XML documents and Java objects
- approach
  - Java classes and a corresponding XML schema
    - classes can be generated from the schema or vice-versa
  - usage
    - creating Java object from XML (unmarshaling)
    - working with the Java objects
    - storing the Java objects to XML (marshaling)

- deprecated since Java 9
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:jxb="http://java.sun.com/xml/ns/jaxb" jxb:version="2.0">
    <xsd:element name="Greetings" type="GreetingListType"/>
    <xsd:complexType name="GreetingListType">
        <xsd:sequence>
            <xsd:element name="Greeting" type="GreetingType" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="GreetingType">
        <xsd:sequence>
            <xsd:element name="Text" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="language" type="xsd:language"/>
    </xsd:complexType>
</xsd:schema>
JAXB – example

- class generation
  - xjc -p hello hello.xsd
public class Hello {

    private ObjectFactory of;
    private GreetingListType grList;

    public Hello() {
        of = new ObjectFactory();
        grList = of.createGreetingListType();
    }

    public void make(String t, String l) {
        GreetingType g = of.createGreetingType();
        g.setText(t);
        g.setLanguage(l);
        grList.getGreeting().add(g);
    }

    public void marshal() {
        try {
            JAXBElement gl = of.createGreetingListType(grList);
            JAXBContext jc = JAXBContext.newInstance("hello");
            Marshaller m = jc.createMarshaller();
            m.marshal(gl, System.out);
        } catch (JAXBException jbe) {
            ...  
        }
    }
}
Hello h = new Hello();
h.make( "Bonjour, madame", "fr" );
h.make( "Hey, you", "en" );
h.marshal();

<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Greetings>
 <Greeting language="fr">
  <Text>Bonjour, madame</Text>
 </Greeting>
 <Greeting language="en">
  <Text>Hey, you</Text>
 </Greeting>
</Greetings>
Scripting API
Overview

- support of scripting languages directly from Java
  - integrating scripts to a Java program
  - calling scripts
  - using Java objects from a script
    • and vice-versa
  - ...
- JSR 223
  - Scripting for the Java™ Platform
- since Java 6 directly part of JDK
  - JavaScript engine is also part of JDK
    - Java 6-7 – Mozilla Rhino engine
    - Java 8 – Nashorn engine
      • an implementation of JavaScript language in Java
      • implements JSR 223
  - there are many implementations for other languages
    • just put a JAR package to the CLASSPATH
Why

- a unified interface for all scripting languages
  - previously, every implementation has its own interface
- easy usage of scripting languages
  - variable “without” types
  - automatic conversions
  - ...
  - no need to compile programs
    - a “shell” can be used
- usage
  - complex configuration files
  - an interface for the application admin
  - extending an application (plugins)
  - scripting in an applications
    - as JS in a browser, VBScript in Office,...
Usage

• package javax.scripting
• ScriptEngineManager
  – a core class
  – obtaining an instance of a script engine
• basic usage
  – an instance of ScriptEngineManager
  – obtaining a particular engine
  – running a script using the eval() method
public class Hello {
    public static void main(String[] args) {
        ScriptEngineManager manager =
            new ScriptEngineManager();
        ScriptEngine engine =
            manager.getEngineByName("JavaScript");
        //ScriptEngine engine =
        //    manager.getEngineByExtension("js");
        //ScriptEngine engine =
        manager.getEngineByMimeType("application/javascript");
        try {
            engine.eval("println( "Hello World!" );");
            System.out.println(
                engine.eval( " 'Hello World again!' ");
        } catch(ScriptException e) { ... }
    }
}
Overview

- **script**
  - a String or char stream (a reader)
  - evaluation via ScriptEngine.eval()
- **interface Compilable**
  - its implementation is optional
    - has to be tested – instanceof Compilable
  - a compilation of a script into byte-code
- **interface Invocable**
  - its implementation is optional
    - has to be tested – instanceof Invocable
    - calling methods and functions of a script
- **Bindings, ScriptContext**
  - environment for script execution
    - mapping variables shared between Java and a script
Obtaining an engine

- ScriptEngineManager.getEngineFactories()
  - a list of all ScriptEngineFactory

```java
for (ScriptEngineFactory factory : engineManager.getEngineFactories()) {
    System.out.println("Engine name: " + factory.getEngineName());
    System.out.println("Engine version: " +
                       factory.getEngineVersion());
    System.out.println("Language name: " +
                       factory.getLanguageName());
    System.out.println("Language version: " +
                       factory.getLanguageVersion());
    System.out.println("Engine names:");
    for (String name : factory.getNames()) {
        System.out.println("  " + name);
    }
    System.out.println("Engine MIME-types:");
    for (String mime : factory.getMimeTypes()) {
        System.out.println("  " + mime);
    }
}
```
Obtaining an engine

- `ScriptEngineFactory.getEngine()`

- or directly

- `ScriptEngineManager.getEngineByName()`
- `ScriptEngineManager.getEngineByExtension()`
- `ScriptEngineManager.getEngineByMimeType()`
Scripts

• evaluating a script
  – Object ScriptEngine.eval( String s, ...)
  – Object ScriptEngine.eval( Reader r, ...)

• passing variables (a basic variant)
  – void ScriptEngine.put(String name, Object value)
  – Object ScriptEngine.get(String name)
  – WARNING: be aware of type conversions
Passing variables

- interface Bindings
  - extends Map<String, Object>
  - a basic implementation – SimpleBindings
- interface ScriptContext
  - an environment, in which scripts run
  - a basic implementation – SimpleScriptContext
  - contains scopes
    - scope = Binding
  - special scopes
    - ENGINE_SCOPE – local for ScriptEngine
    - GLOBAL_SCOPE – global for EngineManager
  - getAttribute(..) / setAttribute(..) corresponds to getBindings(..).get / put
  - std Reader and Writer (input/output) for a script can be set
Passing variables

Calling functions/methods

- interface Invocable
  - optional, has to be tested (instanceof)
  - offers
    - calling script functions from Java code
    - calling script objects' methods from Java code (in a case of object oriented script)
    - implementing a Java interface by functions (methods) of a script

```java
ScriptEngine engine = manager.getEngineByName("javascript");
Invocable inv = (Invocable) engine;

engine.eval("function run() { println( 'function run'); };");
Runnable r = inv.getInterface(Runnable.class);
(new Thread(r)).start();

engine.eval("var runobj = { run: function() 
    { println('method run'); } };");
o = engine.get("runobj");
r = inv.getInterface(o, Runnable.class);
(new Thread(r)).start();
```
• some functions removed (or substituted)
  − mostly from security reasons
• integrated functions for import of Java packages
  − importPackage(), importClass()
    • packages accessible via Packages.PackageName, shortcuts (variables) defined for the most used packages: java (equivalent to Packages.java), org, com,...
    • java.lang is not imported automatically (possible conflicts of objects Object, Math,...)
    • in Java 8 it is necessary to first use
      load("nashorn:mozilla_compat.js");
  − JavaImporter object
    • for “hiding” imported elements to variables (to avoid conflicts)
      var imp = new JavaImporter( java.lang, java.io);
JavaScript engine in JDK (2)

- Java objects in js
  - creating as in Java
  - `var obj = new Clazz( ...)`

- Java arrays in js
  - created via Java reflection
  - `var arr = java.lang.reflect.Array.newInstance( ..)`
  - then used commonly: `arr[i], arr.length,...`

  ```javascript
  var a = java.lang.reflect.Array.newInstance( java.lang.String, 5);
  a[0] = "Hello"
  ```

- anonymous classes
  - anonymous implementation of a Java interface

  ```javascript
  var r  = new java.langRunnable() {
    run: function()
    {
      println( "running...");
    }
  };
  var th = null;
  th = new java.lang.Thread( r); 
  th.start();
  ```
• anonymous classes (cont.)
  - auto-conversion of a function to an interface with a single method

```java
function func() {
    print("I am func!");
}

th = new java.lang.Thread( func);
th.start();
```
overloaded Java methods
- reminder
  overloading “resolved” at compile time (javac)
- when JavaScript variables passed to Java methods, the script engine selects the right variant
- selection can be influenced
  • object[“method_name(parameter_types)”](parameters)
  • warning! string without spaces!
Other engines

• many existing engines
  – awk, Haskell, Python, Scheme, XPath, XSLT, PHP,…
  – usage
    • get JAR
    • put JAR to CLASSPATH
• creating own engine
  – implementing API
    • nutno implementovat alespoň
      – ScriptEngineFactory
      – ScriptEngine
  – zabalení do JAR
    • přidat soubor
      META-INF/services/javax.script.ScriptEngineFactory
      – ScriptEngineManager používá ServiceLoader