Data programmability

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau
Motivating data programmability scenarios
Scenario 1: Store „company“ data in **XML** for platform independent data exchange and human consumption

```xml
<?xml version="1.0" encoding="utf-8" ?>
<company xmlns="http://www.company.softlang.org" >
  <name>ACME Corporation</name>
  <department>
    <name>Research</name>
    <manager>
      <name>Craig</name>
      <address>Redmond</address>
      <salary>123456</salary>
    </manager>
    <employee>
      <name>Erik</name>
      <address>Utrecht</address>
      <salary>12345</salary>
    </employee>
  </department>
  ...
</company>
```

**Q1**: What APIs to use to read, write, and process such XML data?

**Q2**: How to validate XML data in a schema-based manner?

**Q3**: Whether and how to map such data to a domain-specific object model?

Today’s focus is on Q1, less so on Q2. A future lecture will deal with Q3.
Scenario 2: Retrieve „poll“ data from a Polls App through a **Web API** to provide a bot or GUI on top.

What's the coolest language?

- Cobol -- 5 votes
- Python -- 93 votes
- Haskell -- 97 votes
- Java -- 102 votes

Today’s focus is on Q1. A future lecture will deal with Q2 and Q3.

**Q1**: How to represent data for communication between server and client?

**Q2**: How to issue requests and handle responses on a client?

**Q3**: How to set up the API on the server?
Data programmability
— Introduction —
Where is your data?

- **In memory**
  - in variables (primitive values)
  - in structs
  - in trees (algebraic data types)
  - in graphs (objects)

- **Not in memory**
  - in trees (e.g., XML and JSON)
  - in tables (RDBMS)
  - in graphs (e.g., EMF)
  - in sequential files
  - in indexed files
  - or elsewhere (e.g., NoSQL)
Major data paradigms

• **Trees**
  • XML trees
  • JSON trees
  • Algebraic trees

• **Tables**
  • Relational tables in SQL

• **Graphs**
  • OO graphs
  • Model graphs
Major data paradigms:

**trees** vs. **tables** vs. **graphs**

```
<department>
  <name>Research</name>
  <manager>
    <name>Craig</name>
    <address>Redmond</address>
    <salary>123456</salary>
  </manager>
  <employee>
    <name>Erik</name>
    <address>Utrecht</address>
    <salary>12345</salary>
  </employee>
  <employee>
    <name>Ralf</name>
    <address>Koblenz</address>
    <salary>1234</salary>
  </employee>
</department>
```

XML for a company

Major data paradigms:

**trees** vs. tables vs. graphs

```json
{
    "name": "Research",
    "manager": {
        "name": "Craig",
        "address": "Redmond",
        "salary": 123456
    },
    "employees": [
        {
            "name": "Erik",
            "address": "Utrecht",
            "salary": 12345
        },
        {
            "name": "Ralf",
            "address": "Koblenz",
            "salary": 1234
        }
    ]
}
```

Major data paradigms: trees vs. **tables** vs. graphs

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Salary</th>
<th>Manager</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Craig</td>
<td>Redmond</td>
<td>123456</td>
<td>TRUE</td>
<td>42</td>
</tr>
<tr>
<td>2</td>
<td>Erik</td>
<td>Utrecht</td>
<td>12345</td>
<td>FALSE</td>
<td>42</td>
</tr>
<tr>
<td>3</td>
<td>Ralf</td>
<td>Koblenz</td>
<td>1234</td>
<td>FALSE</td>
<td>42</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Relational tables for a company

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Department</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>42</td>
<td>Research</td>
<td>NULL</td>
<td>88</td>
</tr>
<tr>
<td>43</td>
<td>Development</td>
<td>NULL</td>
<td>88</td>
</tr>
<tr>
<td>44</td>
<td>Dev1</td>
<td>43</td>
<td>88</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>88</td>
<td>ACME Corporation</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Major data paradigms: trees vs. tables vs. graphs

Object graph of a company

:Company

<table>
<thead>
<tr>
<th>Name</th>
<th>ACME Corporation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Departments</td>
<td>[o, …]</td>
</tr>
</tbody>
</table>

:Department

<table>
<thead>
<tr>
<th>Name</th>
<th>Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Departments</td>
<td>[…]</td>
</tr>
<tr>
<td>Employees</td>
<td>[o, o, o]</td>
</tr>
</tbody>
</table>

:Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>Craig</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Redmond</td>
</tr>
<tr>
<td>Salary</td>
<td>123456</td>
</tr>
<tr>
<td>Manager</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

:Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>Erik</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Utrecht</td>
</tr>
<tr>
<td>Salary</td>
<td>12345</td>
</tr>
<tr>
<td>Manager</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

:Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>Ralf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>Koblenz</td>
</tr>
<tr>
<td>Salary</td>
<td>1234</td>
</tr>
<tr>
<td>Manager</td>
<td>FALSE</td>
</tr>
</tbody>
</table>
A programming paradigm is a fundamental style of computer programming, a way of building the structure and elements of computer programs. Capabilities and styles of various programming languages are defined by their supported programming paradigms; some programming languages are designed to follow only one paradigm, while others support multiple paradigms.
From data paradigms to technological spaces

What’s a technological space?

Technological space
= Technology and community context in software engineering

- **UMLware** — Mainstream software modeling
- **MDEware** — Model Driven Engineering
- **Javaware** — Mainstream OO programming
- **XMLware** — Interoperability in data exchange
- **SQLware** — Mainstream database management
- **Pythonware** — Scripting and frameworks
- **RDFware** — Semantic Web and Linked Data
- **JSONware** — Modern interoperability
- **COBOLware** — Legacy programming technology
Technological Spaces: an Initial Appraisal

Ivan Kurtev ¹, Jean Bézivin ², Mehmet Aksit ¹

¹ Software Engineering Group (TRESE), University of Twente, The Netherlands
{kurtev, aksit}@cs.utwente.nl

² Faculty of Sciences, University of Nantes, France
bezivin@sciences.univ-nantes.fr

Abstract. In this paper, we propose a high level view of technological spaces (TS) and relations among these spaces. A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference regular meetings. Although it is difficult to give a precise definition, some TSs can be easily identified, e.g. the XML TS, the DBMS TS, the abstract syntax TS, the meta-model (OMG/MDA) TS, etc. The purpose of our work is not to define an abstract theory of technological spaces, but to figure out how to work more efficiently by using the best possibilities of each technology. To do so, we need a basic understanding of the similarities and differences between various TSs, and also of the possible operational bridges that will allow transferring the results obtained in one TS to other TS. We hope that the presented industrial vision may help us putting forward the idea that there could be more cooperation than competition among alternative technologies. Furthermore, as the spectrum of such available technologies is rapidly broadening, the necessity to offer clear guidelines when choosing practical solutions to engineering problems is becoming a must, not only for teachers but for project leaders as well.
Technological Spaces: an Initial Appraisal

Ivan Kurtev 1, Jean Bézivin 2, Mehmet Aksit 1

1 Software Engineering Group (TRESE), University of Twente, The Netherlands
{kurtev, aksit}@cs.utwente.nl
2 Faculty of Sciences, University of Nantes, France
bezivin@sciences.univ-nantes.fr

Abstract. In this paper, we propose a high level view of technological spaces (TS) and relations among those spaces. A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared experiences, educational support, and can be naturally combined with conferences or meetings. Although it is difficult to give a precise definition, some TSs can be easily identified, e.g. the XML FDA, the UML TS, the business syntax TS, the XML-RDF TS, etc. The purpose of our work is not to define a single theory of technological spaces, but to figure out how to work more efficiently by using the best possibilities of each technology. To do so, we need a basic understanding of the similarities and differences between various TSs, and some of the possible operational changes that will allow transferring the results obtained in one TS to other TS. We hope that the presented industrial vision may help us putting forward the idea that there could be more cooperation than competition among alternative technologies. Furthermore, as the spectrum of such available technologies is rapidly broadening, the necessity to offer clear guidelines when choosing practical solutions to engineering problems is becoming a must, not only for teachers but for project leaders as well.
More technological spaces ...

Data driven journalism

CSVware

Specware

Javaware

???

Lambdaware

Rubyware
Constituents of a technological space

• Application scenarios
• Programming languages
• Data representation formats
• Development tools such as IDEs and compilers
• Runtime tools such as libraries and frameworks
• Query and transformation languages
• Textbooks and other knowledge resources
• Conferences and other community resources
• …
Space exploration with 101companies

101companies: a community project on software technologies and software languages by Jean-Marie Favre, Ralf Lämmel, Thomas Schmorleiz, Andrei Varanovich.

http://101companies.org/
Back to data paradigm:

**Constituents of a data paradigm**

- The basic paradigm
  - A data model
  - A type system
  - Dedicated query and transformation languages

- The **programmability** part
  - In-memory representations
  - De-/serialization semantics
  - APIs for data access, query, and transformation
  - Mapping approaches (not covered in today’s lecture)
XMLware
XML basics
Classic motivation of XML

Consider the following realities:
- Applications are built in Cobol, C#, Java, Python, etc.
- “Messages” have to be send over the “wire”.
- Configuration data has to be represented somehow.
- Objects have to be persisted in an human-readable way.

Conclusions:
- We need to model data in a language-independent manner.
- We need a comprehensive method for exchange/storage formats.

We need (something like) XML.
XML stands for eXtensible Markup Language.
Perhaps, JSON would also do. Let’s discuss this later.
Sample XML document

From: XML: A Primer, by Simon St. Laurent

```xml
<?xml version="1.0"?>
<weatherReport>
  <date>7/14/97</date>
  <city>North Place</city>, <state>NX</state>
  <country>USA</country>
  High Temp: <high scale="F">103</high>
  Low Temp: <low scale="F">70</low>
  Morning: <morning>Partly cloudy, Hazy</morning>
  Afternoon: <afternoon>Sunny & hot</afternoon>
  Evening: <evening>Clear and Cooler</evening>
</weatherReport>
```
Another sample XML document

```xml
<?xml version="1.0" encoding="utf-8" ?>
<company xmlns="http://www.company.softlang.org">
  <name>ACME Corporation</name>

  <department>
    <name>Research</name>
    <manager>
      <name>Craig</name>
      <address>Redmond</address>
      <salary>123456</salary>
    </manager>
  </department>

  <employee>
    <name>Erik</name>
    <address>Utrecht</address>
    <salary>12345</salary>
  </employee>

  ...
```
XML-based languages
- Some examples -

- FpML – Financial products Markup Language
- HL7 – Standards in Healthcare
- MathML – Mathematical Markup Language
- XHTML – HTML defined in XML
- … and countless others …
Well-formedness of XML

- A document has exactly one root element.
- There is an end tag for each start tag: `<name>….</name>`
- Empty elements can be abbreviated: `<noname/>`
- Elements must be properly nested.
- Attribute values must be enclosed into “…”.
- …
XML-related technologies and standards

• XML is a standard; there is 1.0 and 1.1.
• XML elements & attributes are qualified by XML namespaces.
• DTD (Document Type Definition; part of XML standard) and XML Schema and Relax NG (both separate standards) are used to define legal XML tags and their attributes for particular XML languages.
• CSS (Cascading Style Sheets) and XSL (eXtensible Stylesheet Language) are used to map HTML or XML for presentation in a browser.
• XPath, XSLT and XQuery are used to program queries on transformations on XML data. In fact, XPath is part of XSLT and XQuery.
• DOM (Document Object Model), SAX (Simple API for XML), and JAXP (Java API for XML Processing) are all Java APIs for XML processing. In fact, DOM is language-independent.
XML queries with XPath

• XPath is an XML query language.
• Queries are composed from query axes.
  • Children, Ancestors, Descendants, ...
• XPath is embedded into Java et al. by APIs.
  • XPath queries are encoded as strings.
• Examples:
  “//salary”
  “//manager/salary”

All salary nodes below all manager nodes anywhere
### XPath axes

<table>
<thead>
<tr>
<th>AxisName</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>Selects all ancestors (parent, grandparent, etc.) of the current node</td>
</tr>
<tr>
<td>ancestor-or-self</td>
<td>Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself</td>
</tr>
<tr>
<td>attribute</td>
<td>Selects all attributes of the current node</td>
</tr>
<tr>
<td>child</td>
<td>Selects all children of the current node</td>
</tr>
<tr>
<td>descendant</td>
<td>Selects all descendants (children, grandchildren, etc.) of the current node</td>
</tr>
<tr>
<td>descendant-or-self</td>
<td>Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself</td>
</tr>
<tr>
<td>following</td>
<td>Selects everything in the document after the closing tag of the current node</td>
</tr>
<tr>
<td>following-sibling</td>
<td>Selects all siblings after the current node</td>
</tr>
<tr>
<td>namespace</td>
<td>Selects all namespace nodes of the current node</td>
</tr>
<tr>
<td>parent</td>
<td>Selects the parent of the current node</td>
</tr>
<tr>
<td>preceding</td>
<td>Selects everything in the document that is before the start tag of the current node</td>
</tr>
<tr>
<td>preceding-sibling</td>
<td>Selects all siblings before the current node</td>
</tr>
<tr>
<td>self</td>
<td>Selects the current node</td>
</tr>
</tbody>
</table>
XML transformations with XSLT

• XSLT is an XML transformation language.

• XSLT leverages XPath for node selection.

• XSLT is an XML-based language.

• XSLT is a functional programming language.
Cutting salaries with XSLT

```xml
<xsl:stylesheet>
  <xsl:template match="salary">
    <xsl:copy>
      <xsl:value-of select=". div 2"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

Type-specific template

Generic default

Recursion into kids
XML languages
(defined by XML schemas)
Motivation of XML schemas

OK

<employee>
  <name>Ralf</name>
  <address>Koblenz</address>
  <salary>1234</salary>
</employee>

Not OK!

<employee>
  <name>Ralf</name>
  <salary>1234 Euro</salary>
</employee>

Address missing

Not a number
XML Schema (XSD)

http://www.w3.org/XML/Schema:

“XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content and semantics of XML documents. […] XML Schema was approved as a W3C Recommendation on 2 May 2001.”
XSD for 101

```xml
<xs:element name="company">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element maxOccurs="unbounded" minOccurs="0" ref="department"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="department">

<xs:complexType name="employee">

```
XML validation

- Well-formed XML as a prerequisite

- Input
  - XML document
  - XML schema

- Output
  - “valid”: document valid w.r.t. schema
  - “invalid” + violations
XSD – schema components

- Element declarations
- Complex type definitions
- Model-group definitions
- Simple type definitions
- Attribute declarations
- Attribute-group definitions
- Redefinitions
- Annotations

Sets of XML trees rooted by a certain element name
Recursive macros with subtyping
Nonrecursive macros without subtyping
Types of leafs in XML trees (both elements and attributes).
Deprecated
Comments and hints for schema processors
Sketch of the company schema

<xs:schema ...

  <xs:element name="company"> ... </xs:element>
  <xs:element name="department"> ... </xs:element>
  <xs:complexType name="employee"> ... </xs:complexType>
  <xs:element name="name"> ... </xs:element>
  <xs:element name="address"> ... </xs:element>
  <xs:element name="salary"> ... </xs:element>

</xs:schema>
An employee element has children for name, address, and salary.
In a variation of our preferred schema, a subunit (of a department) is either an employee or a department.
“∗” (i.e., many)

```xml
<xs:element name="company">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element maxOccurs="unbounded"
                   minOccurs="0"
                   ref="department"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

A company element has any number of department elements as its children.
XSD simple types

• Comparable to primitive types in a programming.
• Example:
  
  <xs:element name="salary" type="xs:double"/>

• All attributes are of simple types.

• New simple types can be defined by:
  – Restriction
  – Union
  – List
Options for XML processing in an OO programming language

- APIs for push-based parsers
  - Java’s **SAX**, ...
- APIs for pull-based parsers
  - .NET’s XmlReader, Java’s StAX, ...
- APIs for in-memory XML trees
  - W3C’s **DOM**, Java’s JDOM, .NET’s LinqToXml
- Programming languages with XML support
  - VB.NET, ...
The DOM option
(DOM=Document Object Model)

Source: Armstrong: "Working with XML"
What’s DOM?

• An **object model** for XML trees.

• **Central types:**
  – Document
  – Attribute
  – Node
    • Element
    • Text
    • Comment
    • CData

• **API segments:**
  – Construction
  – Navigation
  – Modification
Feature „Total“ with the JDOM API

http://101companies.org/wiki/Contribution:jdom

public class Total {

    public static double total(Document doc) {

        // Aggregate salaries
        double total = 0;

        // Iterate over all salary elements
        Iterator<?> iterator = doc.getDescendants(new ElementFilter("salary"));
        while (iterator.hasNext()) {
            Element elem = (Element)iterator.next();
            Double salary = Double.valueOf(elem.getText());
            total += salary;
        }

        return total;
    }

}
Feature „Cut“ with the JDOM API

http://101companies.org/wiki/Contribution:jdom

```java
public class Cut {

    public static void cut(Document doc) {

        // Iterate over all salary elements
        Iterator<? extends Element> iterator = doc.getDescendants(new ElementFilter("salary"));

        // Snapshot these elements before modification
        List<Element> elems = new LinkedList<Element>();
        while (iterator.hasNext())
            elems.add((Element)iterator.next());

        // Iterate over salary elements and cut salaries
        for (Element elem : elems) {
            Double salary = Double.valueOf(elem.getText());
            elem.setText(Double.toString(salary/2));
        }
    }
}
```

DOM objects are mutable.
The SAX option
(SAX=Simple API for XML)

Source: Armstrong: “Working with XML”
What’s SAX?

• A **framework** for event handling-based XML parsers.
• **Typical events**
  • Open element
  • Close element
  • Find text
  • ...

Feature „Total“ with the SAX API
http://101companies.org/wiki/Contribution:sax

Event handler for „start“ element: determine whether „salary“ element begins.

```java
/**
 * Handle "start element"
 */
public void startElement(
    String uri,
    String name,
    String qName,
    Attributes atts) {
    isSalary = (uri.equals(ns) && name.equals("salary"));
}
```

See the complete code online.
Feature „Cut“ with the SAX API

http://101companies.org/wiki/Contribution:sax

```java
public void startElement(String uri, String name, String qName,
    Attributes attrs) {

    try {
        isSalary = (uri.equals(ns) && name.equals("salary"));
        writer.writeStartElement(name);
        if (!uri.equals(lastNamespace)) {
            lastNamespace = uri;
            writer.setDefaultNamespace(uri);
            writer.writeDefaultNamespace(uri);
        }
        for (int i = 0; i < attrs.getLength(); i++) {
            writer.writeAttribute(attrs.getLocalName(i), attrs.getValue(i));
        }
    } catch (XMLStreamException e) {
        e.printStackTrace();
    }
}
```

Copy input to output stream with extra „writer“ API.

See the complete code online.
JSONware
Store „company“ data in JSON for platform independent data exchange and human consumption

So we use JSON rather than XML this time around.

```json
{
  "name" : "Acme Corporation",
  "departments" : [
    {
      "name" : "Research",
      "manager" : {
        "name" : "Fred",
        "salary" : 88888
      }
    },
    {
      "name" : "Development",
      ...
    }
  ]
}
```
JSON — The JavaScript Object Notation

http://101companies.org/wiki/Language:JSON

• Inspired by JavaScript syntax and type system
  • Objects are essentially dictionaries.
    • **Keys** are strings. **Values** are JavaScript values.
• Thought to be much simpler than XML
• Very well supported across many platforms
• Increasingly dominant in the Web context
JSON syntax
JSON syntax I/III

Source: http://www.json.org/

© 2010-16 Ralf Lämmel
JSON syntax II/III

Source: http://www.json.org/
JSON syntax III/III

Source: http://www.json.org/
Feature „Total“ with the javax.json API

http://101companies.org/wiki/Contribution:javaJson

```java
package org.softlang.company.features;

import javax.json.JsonArray;
import javax.json.JsonNumber;
import javax.json.JsonObject;
import javax.json.JsonValue;

public class Total {

    /**
     * @param obj
     *    JSON object to traverse
     * @return salary total
     */
    public static double total(JsonObject obj) {
        return total(null, obj);
    }

    /**
     * @param key1
     *    key for the value at hand
     * @param vall
     *    JSON value to traverse
     * @return salary total
     */
    private static double total(String key1, JsonValue vall) {
```

Conceptually, the code for JSON processing is similar to the one for XML processing.

There is several JSON APIs for Java. We use the platform standard here.

See the complete code online.
Feature „Cut“ with the javax.json API
http://101companies.org/wiki/Contribution:javaJson

```java
private static void cut(JsonObjectBuilder builder, String key, JsonValue val) {
    switch (val.getValueType()) {
    case NUMBER:
        Double num = ((JsonNumber) val).doubleValue();
        if (key.equals("salary")) {
            // Halve salary
            num /= 2;
        }
        builder.add(key, num);
        break;
    case OBJECT:
        builder.add(key, cut((JsonObject) val));
        break;
    case ARRAY:
        builder.add(key, cut((JsonArray) val));
        break;
    default:
        builder.add(key, val);
        break;
    }
}
```

The javax.json API assumes **immutable** JSON values. Thus, transformations need to **copy**.

See the complete code online.
JSON schema

http://json-schema.org/examples.html
SQLware
Elevator speech

Think of information systems and data processing!

1. How to persist data?
2. How to separate data and functionality?
3. How to deal with a lot of data efficiently?
4. How to implement entity relationships?

XML and JSON may serve 1.-2.
Relational databases serve 1.-4.

Exercise: what’s XML specifically good for?

Also: how to remain an OO programmer?
Database programming (ignoring OO specifics)

1. Define tables according to relational model
2. Implement relational model via SQL
3. Implement CRUD functionality

Tables = rows / columns of cells

Create, Read, Update, Delete
The relational model
Relations (tables)

- Relation
  - Vertically: set of tuples (“rows”)
  - Horizontally: set of columns
- Each cell is of some type
  - Strings
  - Numbers
  - Row IDs (numbers again)
Relational algebra: compute relations

- **Projection** (select columns)
- **Selection** (select rows)
- **Join** (compose two tables by condition)
### Projection

(select columns)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>salary</th>
<th>manager</th>
<th>cid</th>
<th>did</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Craig</td>
<td>Redmond</td>
<td>30864</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Ray</td>
<td>Redmond</td>
<td>58641.75</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Klaus</td>
<td>Boston</td>
<td>5864</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Karl</td>
<td>Riga</td>
<td>586.25</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Erik</td>
<td>Utrecht</td>
<td>3086.25</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Ralf</td>
<td>Koblenz</td>
<td>308.5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Joe</td>
<td>Wifi City</td>
<td>586</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Selection
(select rows)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>salary</th>
<th>manager</th>
<th>cid</th>
<th>did</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Craig</td>
<td>Redmond</td>
<td>30864</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Ray</td>
<td>Redmond</td>
<td>58641.75</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Klaus</td>
<td>Boston</td>
<td>5864</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Karl</td>
<td>Riga</td>
<td>586.25</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Erik</td>
<td>Utrecht</td>
<td>3086.25</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Ralf</td>
<td>Koblenz</td>
<td>308.5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Joe</td>
<td>Wifi City</td>
<td>586</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>
Join
(compose two tables by condition)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>address</th>
<th>salary</th>
<th>manager</th>
<th>cid</th>
<th>did</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Craig</td>
<td>Redmond</td>
<td>30864</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Ray</td>
<td>Redmond</td>
<td>58641.75</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Klaus</td>
<td>Boston</td>
<td>5864</td>
<td>1</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Karl</td>
<td>Riga</td>
<td>586.25</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>Erik</td>
<td>Utrecht</td>
<td>3086.25</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>Ralf</td>
<td>Koblenz</td>
<td>308.5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Joe</td>
<td>Wifi City</td>
<td>586</td>
<td>0</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME Corporation</td>
</tr>
<tr>
<td>ACME Corporation</td>
</tr>
<tr>
<td>ACME Corporation</td>
</tr>
<tr>
<td>ACME Corporation</td>
</tr>
<tr>
<td>ACME Corporation</td>
</tr>
</tbody>
</table>
Relational schemas

Key terms

- Attributes (names)
- Attribute domains (types)
- Relational schema (attribute-domain pairs)
- Instance of relational schema (sets of tuples)
The relational schema for 101 companies

- Relational schemas (names only)
  - company (id, name)
  - department (id, name, cid, did)
  - employee (id, name, address, salary, manager, cid, did)

Key constraints:
Primary key (underlined) for identification of tuple
Foreign key (italics) for reference to another table
**Variation**

- Relational schemas (names only)
  - company (id, name)
  - department (id, name, cid, did)
  - employee (id, name, address, salary, manager, cid, did)

**Key constraints:**
Primary key (underlined) for identification of tuple
Foreign key (italics) for reference to another table
Variation

Relational schemas (names only)

- company (id, name)
- department (id, name, cid, did, mid)
- employee (id, name, address, salary, manager, cid, did)

Key constraints:
Primary key (underlined) for identification of tuple
Foreign key (italics) for reference to another table
Database programming with SQL (Structured Query Language)

- Represent schema in DDL subset of SQL
  - DDL - Data Definition Language
    - Part of SQL for data definition
- Represent population in DML subset of SQL
  - DML - Data Manipulation Language
    - Part of SQL for CRUD (Create, Read, Update, Delete)
CREATE TABLE company (  
id INTEGER,  
name VARCHAR(100)  
)
CREATE TABLE company (  
id INTEGER PRIMARY KEY,  
name VARCHAR(100) UNIQUE NOT NULL  
)

CREATE TABLE company (  
id INTEGER AUTO_INCREMENT PRIMARY KEY,  
name VARCHAR(100) UNIQUE NOT NULL  
)
CREATE TABLE department (
    id INTEGER,
    name VARCHAR(100),
    cid INTEGER,
    did INTEGER,
)

CREATE TABLE department (  
id INTEGER PRIMARY KEY,  
name VARCHAR(100) NOT NULL,  
cid INTEGER NOT NULL,  
did INTEGER,  
FOREIGN KEY (cid) REFERENCES company(id),  
FOREIGN KEY (did) REFERENCES department(id))
CREATE TABLE department (  
id INTEGER PRIMARY KEY,  
name VARCHAR(100) UNIQUE NOT NULL,  
cid INTEGER NOT NULL,  
did INTEGER,  
FOREIGN KEY (cid) REFERENCES company(id)  
  ON DELETE CASCADE ON UPDATE CASCADE,  
FOREIGN KEY (did) REFERENCES department(id)  
  ON DELETE CASCADE ON UPDATE CASCADE  
)

CREATE TABLE employee (  
id INTEGER PRIMARY KEY,  
name VARCHAR(50) NOT NULL,  
address VARCHAR(50) NOT NULL,  
salary DOUBLE NOT NULL,  
manager BOOL NOT NULL,  
cid INTEGER NOT NULL,  
did INTEGER NOT NULL,  
FOREIGN KEY (cid) REFERENCES company(id),  
FOREIGN KEY (did) REFERENCES department(id)  
)
DDL language summary

- CREATE TABLE
  - INTEGER, VARCHAR(...), DOUBLE
  - NOT NULL
  - UNIQUE
  - PRIMARY / FOREIGN KEY
  - ON DELETE / UPDATE CASCADE
SQL DML
CRUD

- C: Create (SQL: Insert)
- R: Read (SQL: Select)
- U: Update
- D: Delete
INSERT INTO company (name) 
VALUES ("Acme Corporation")

Insert a new company into the corresponding table.
INSERT INTO department (name,cid) VALUES ("Research",1)
INSERT INTO department (name,cid) VALUES ("Development",1)
...

Insert several departments into the corresponding table.
SELECT * FROM DEPARTMENT

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>cid</th>
<th>did</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Research</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>Development</td>
<td>1</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Dev1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>Dev1.1</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

List of tuples of the department table.
SELECT SUM(salary) FROM employee

Select all employees, project to their salaries, and sum them up.
SELECT SUM(salary) FROM employee
WHERE cid = 1

Retrieve only salaries of a specific company.
SELECT SUM(salary) FROM employee
WHERE cid =
(SELECT id FROM company
 WHERE name = "Acme Corporation")

Use a nested query to determine the company id.
UPDATE employee
SET salary = salary / 2

Cut all salaries in half.
UPDATE employee
SET salary = salary / 2
WHERE cid = 1

Limit update to employees with company id = 1.
UPDATE employee
SET salary = salary / 2
WHERE cid =
(SELECT id FROM company
WHERE name = "Acme Corporation")

Use a nested query to determine the company id.
http://101companies.org/wiki/Contribution:mySqlMany

We use a local database server and SQL monitor; see the online documentation for the contribution.
Embedding SQL
with JDBC

http://101companies.org/wiki/Contribution:simplejdbc
public static double total(MyConnection myConnection, String companyName) {
    double total = 0;
    try {
        String query = "SELECT salary FROM employee 
                        + "WHERE cid = (SELECT id FROM company WHERE name = ?);";
        PreparedStatement pstmtEmployees = myConnection.getConn().prepareStatement(query);
        pstmtEmployees.setString(1, companyName);
        ResultSet salaries = pstmtEmployees.executeQuery();
        while (salaries.next())
            total += salaries.getDouble("salary");
    } catch (SQLException e) {
        e.printStackTrace();
    }
    return total;
}
Library support for database programming

- JDBC (part of Java Core API)
  - Connections to databases
  - Submit SQL statements
  - Retrieve results
- MySQL connector (part of demo project)
  - JDBC-based driver for MySQL
Embedded SQL

- Important JDBC types
  - Connection
  - Statement (different forms thereof)
  - ResultSet (for queries in particular)
  - SQL Exception
public static void cut(MyConnection myConnection, String companyName) {
    try {
        // cut salaries in all employee columns
        String sqlCut = "UPDATE employee SET salary = salary / 2 "
                        + "WHERE cid = (SELECT id FROM company WHERE name = ?);";
        PreparedStatement pstmtEmployees = myConnection.getConn()
                                        .prepareStatement(sqlCut);
        pstmtEmployees.setString(1, companyName);
        pstmtEmployees.executeUpdate();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

Exercise: understand the notion of injection attacks and argue how “prepared statements” help avoiding the problem.
End of Lecture