# Serialisierung

**Beschreiben Sie das Prinzip der Serialisierung?**

Die Serialisierung eines Objekts ist die Umwandlung der im Hauptspeicher vorliegenden Objektdaten in eine Form, die es erlaubt diese Daten in eine Datei zu schreiben oder über eine Netzverbindung zu transportieren. Dabei muss das Objekt aus diesen umgewandelten Daten im Speicher regenerierbar sein.

* Deserialisierung – Rückumwandlung
* Umwandlung eines Objektes von Hauptspeicher
* Nach Umwandlung liegt ein Byte-Datenstrom vor
* Persistente Speicherung auf HDD oder Transport für Netzwerk

**Was versteht man unter dem Begriff Marshalling im Kontext der Serialisierung?**

* Unter dem Begriff „Marshalling“ versteht man den Prozess der Transformation einer beliebigen Datenstruktur in eine übertragbare Nachricht/Format:
	+ „planieren“ der Datenstruktur (in eine zusammenhängende Nachricht)
	+ Übersetzung in das gemeinsame Format

Konkret: Umwandlung von strukturierten Daten in ein anderes Format

* Anwendung: Datenumwandlung unterschiedlicher Technologien und ggf. auch Programmiersprache
	+ Bsp.: OR-Mapper  Objekttransformation auf Datenbanken
	+ XML Marshalling in JavaFX
	+ XML Marshalling in C# in WPF

**Welche Elemente benötigt man für eine Serialisierung?**

* Instanz einer Klassen
* Klasse muss Serializable implementiert haben
* private static final long serialVersionUID = 20080330L;
* FileInputStream oder InputStream
* FileOutputStream oder OutputStream
* ObjectInputStream
	+ Casten von Objekt zur Ausgangsklasse
* ObjectOutputStream

**Nachteil der Serialisierung?**

* Aufwändig, da alle Strukturdaten mitgespeichert werden
* Der Bytecode der zu deserialisierenden Klasse muss vorhanden sein

# Interprozess Kommunikation (IPC)

**Welches sind die wichtigsten Komponenten von IPC?**

* Synchronisation (globale Semaphore)
* Kommunikation
	+ Gemeinsamer Speicher
	+ Nachrichtenbasiert

**Nennen Sie Beispiele für prozessübergreifende Kommunikation?**

* Maustaste drücken
* Taste drücken
* Grafikausgabe
* Java-Pipes, normale Pipes,
	+ Linux-Pipes: ls | sort

**Welche Komponenten gibt es für die Kommunikation?**

* Send
* Receive
	+ Bockiert
* Pipes in Unix/Linux
* Pipes in Java
	+ **PipedInputStream**
	+ **PipedOutputStream**

**Beschreiben Sie in Stichworten die Kommunikation über Dateien**

* ältester IPC-Mechanismus
* Sender schreibt Daten in Datei
* Empfänger liest Daten aus Datei
* Über ein globales Semaphor/Mutex wird das lesen und Speichern gesteuert.
* nachteilig: zweimaliger Zugriff auf Massenspeicher
* aber: es gibt auch Dateisysteme im RAM
* nachteilig: überlappender Zugriff durch Sender und Empfänger (wenn ohne Semaphor)
* Abhilfe:
	+ Globale Semaphore (in C, C++, Delphi, C#,
	+ Lösung: Sperren der Datei (File Locking), z. B. mittels lockf()
	+ Beispiel: src/vl/filelock.c (extern)
	+ Problem: Sperren setzt Wohlverhalten voraus oder einen Absturz

**Was ist ein Socket?**

* Rechnerübergreifende IPC: Sockets (siehe Server: IP-Adresse+Port)
* Bidirektional
* Gepuffert
* Abstrahiert von Details des Kommunikationssystems
* Beschrieben durch Domäne (Protokollfamilie), Typ und Protokoll
* De-facto-Standard einer API für Kommunikation über das Internetprotokoll(IP)
	+ Sockets dienen als Kommunikationsendpunkte die von Programmen vom Betriebssystem angefordert werden können (Bsp.: Webserver, Chat, DB-Server)
* Es gibt zwei Arten von Sockets:
	+ verbindungslose UDP/IP-Sockets und
	+ verbindungsorientierte TCP/IP-Sockets.
* Sockets ermöglichen sprachunabhängige Interprozesskommunikation
* Aber: Es gibt im IP-Protokollstack keine Datenrepräsentationsschicht
* Daten müssen in sprachunabhängiges Transportformat konvertiert werden
	+ viele Internet-Protokolle sind deshalb textbasiert (MIME)

**Nennen Sie die wichtigsten Sockettypen:**

* stromorientiert, verbindungsorientiert und gesichert
* nachrichtenorientiert und ungesichert
* nachrichtenorientiert und gesichert

**Welche Protokolle gibt es dafür?**

* TCP/IP Protokoll
	+ strom- und verbindungsorientiert, gesichert
	+ Reihenfolge ist wichtig
* UDP/IP Protokoll
	+ nachrichtenorientiert, verbindungslos, ungesichert
	+ Nachrichten können verloren oder dupliziert werden
	+ Reihenfolge kann durcheinander geraten
* Paketgrenzen bleiben erhalten (Datagramm-Protokoll)

**Wie funktioniert die Kommunikation per Server Verbindungsorientiert?**

* Adresse
	+ IP und Port
* Server lauscht
	+ Bei einem Verbindungswunsch erzeugt er einen Kommunikationssocket („accept“)
	+ Der Kommunikationssocket hat eine andere Port-Nummer als der Verbindungswunschsocket! (Port $80, 0x80)

**Zeichnen und beschreiben Sie die zeitliche Reihenfolge bei einer Server-Verbindung auf?**

**Server**

* 4 Aktionen sind notwendig um mit einem Client Daten auszutauschen:
	+ Socket Erzeugung auf dem Server
	+ Socket auf einen lokalen Endpunkt binden
	+ Socket in den “listen,lauschen” Modus versetzen
	+ Ankommende Verbindung akzeptieren

**Client**

* 2 Schritte sind notwendig für einen Client:
	+ Erzeugung des Socket
	+ Verbinde den Socket mit der Remote-Address des Servers + Port



**Was versteht man unter „Marshalling“?**

* Unter „Marshalling“ versteht man den Prozess der Transformation einer beliebigen Datenstruktur in eine übertragbare Nachricht:
	+ „planieren“ der Datenstruktur (in eine zusammenhängende Nachricht)
	+ Übersetzung in das gemeinsame Format (MIME)

**Welche Formate gibt es beim Marshalling?**

* ASN.1 (ISO OSI)
* XDR (Internet RPC)
* CDR (CORBA)
* Java object serialization
* XML
	+ Übertragung in binärer Form
	+ Übertragung als ASCII
* **JSon**
	+ **https://chemnitzer.linux-tage.de/2015/media/vortraege/folien/431-json-rpc-web-api.pdf**
* **SOAP**

**Was ist SOAP?**

## SOAP

Als Weiterentwicklung von XML-RPC gilt das Protokoll *SOAP*, dessen Abkürzung ursprünglich für *Simple Object Access Protocol* steht, in der Zwischenzeit aus rechtlichen Gründen (in den USA können Abkürzungen nicht als Marke eingetragen werden) nicht mehr als Abkürzung gilt, sondern als eigenständiges Kunstwort.

An der Entwicklung von SOAP waren ebenfalls die Protagonisten der XML-RPC-Entwicklung, Dave Winer und das Unternehmen Microsoft beteiligt, die SOAP im Jahre 1999 in ersten Versionen veröffentlichten. An der Entwicklung von SOAP beteiligten sich jedoch nach und nach immer mehr Unternehmen, so dass SOAP sich nach und nach immer stärker etablierte und die Spezifikation der Version 1.1 später beim World Wide Web Consortium (W3C) eingereicht wurde. Daraus bildete sich eine offizielle Arbeitsgruppe, die im Jahr 2003 die SOAP-Version 1.2 veröffentlichte, die als offizielle Empfehlung gilt.

Im Gegensatz zu XML-RPC ist SOAP universeller bei der Auswahl des Übertragungsprotokolls. Neben XML über HTTP kann SOAP Nachrichten beispielsweise auch per FTP oder SMTP übertragen. Zudem bietet SOAP durch seinen flexiblen Aufbau Möglichkeiten, umfangreiche Abfragen innerhalb einer Transaktion zu kombinieren, um den Übertragungs- und Verarbeitungsaufwand zu verringern. In Kauf genommen wird dies jedoch alles durch einen teilweise erheblich komplexeren Aufbau von SOAP-Nachrichten als bei XML-RPC.

Dennoch ist SOAP inzwischen auf dem besten Wege, in der Programmierwelt zu einer der wichtigsten Datenaustauschprotokolle zu werden - nicht zuletzt durch die Entwicklungen im Bereich Web 2.0 (siehe hierzu auch [Web 2.0 - Die nächste Web-Generation](https://www.netplanet.org/www/web20.shtml)).

# RPC Remote Procedure Call

**Beschreiben Sie das Prinzip von RPC:**



* Getrennte Module
* Schnittstellen

**Beschreiben Sie Unterschiede im Datenfluss**



**Eigenschaften RPC?**

* Synchron:
	+ Der Aufrufer wird blockiert, bis der Aufgerufener das Ergebnis abliefert.
* Prozeduraufruf:
	+ Signatur der Prozedur/Methode definiert zu übertragende Daten
* Unterschiedlicher Adressraum:
	+ Speicheradressen (Zeiger) sind nicht semantik-erhaltend übertragbar
* Schmalbandig:
	+ Bandbreite des Netzes ist um Dimensionen geringer als die der Kommunikationspfade innerhalb eines Rechners
	+ Performance !!!
	+ Verhältnis: Übertragungszeit zum Server mit Gesamtserverrechenzeit

**Vorteile von RPC?**

* Bessere Aufgabenverteilung
* Bessere Lastverteilung
* Bessere Ressourcennutzung
* Bessere Modularität
* Bessere Wiederverwendbarkeit
* Größere Offenheit
* Bessere
* Integrationsfähigkeit
* … u. v. m.

**Beispiele für RPC?**

* Anmeldung in einem System (login)
* Middleware (Bestellung, Warenlager)
* Telefonauskunft
* Siri/Cortana

**Fehlertypen in RPC?**

Äußerst wichtig ist bei einer solchen Kommunikation über ein Netzwerk die so genannte Fehlersemantik, also die Reaktion auf Kommunikationsfehler. Dies muss der Programmierer einer Anwendung unbedingt berücksichtigen, um beispielsweise bei einer fehlenden Server-Antwort auf eine RPC-Nachricht diesen "hängenden" Zustand abzufangen. Gleichzeitig muss unbedingt vermieden werden, dass wiederholt gesendete RPC-Nachrichten beim Server auch als solche erkennbar sind, damit dieser Vorgänge nicht doppelt ausführt. Es gibt deshalb in RPC verschiedene Möglichkeiten, solche Zustände und wiederholende Nachrichten zu definieren, die zur Gewährleistung der Datenintegrität von Programmierern auch zwingend genutzt werden.

**Unterschiede in synchronen und asynchronen RPC?**

Unterschieden wird bei der RPC-Verarbeitung zwischen *synchronem RPC* und *asynchronem RPC*. Bei synchronem RPC muss die Verarbeitung genau Schritt für Schritt erfolgen. Das bedeutet, dass der Client eine RPC-Nachricht an den Server sendet und zwingend auf die Antwort warten muss, bevor er weiter fortsetzen kann. Bei Anwendungen, die mit asynchronem RPC kommunizieren, kann der Client nach dem Senden einer RPC-Nachricht weitere Operationen vornehmen, da auch eine späterer Empfang der Server-Antwort einkalkuliert ist. Diese Funktionsweise hat durchaus seine zentrale Bedeutung, denn Anwendungen, die mit synchronem RPC kommunizieren, ist es sehr wichtig, dass die Anwendung eine höhere Laufzeit der Server-Antwort berücksichtigt, da sonst unter Umständen bei höheren Wartezeiten die Anwendung schlicht stehen bleibt und für den Benutzer unangenehme Wartezeiten und der Eindruck eines "hängenden Programms" entsteht (was an sich dann auch tatsächlich der Fall ist). Andererseits erfordern Anwendungen mit asynchronem RPC in der Regel einen höheren Programmieraufwand.

**Ablauf von RPC?**



**Anforderung der Umsetzung?**

* Semantik im Fehlerfall
* Semantik von Zeigern
* Einbettung in Programmiersprachen
* Auffinden und Binden an die entfernten Prozeduren
* Protokoll des Datenaustauschs (Stubs)
* Eigenschaften der Kommunikation

**Zeigen Sie den zeitlicher Ablauf mit den RPC-Komponenten**



# RMI Remote Method Invocation (RPC)

**Voraussetzung von RMI?**

* Java RMI setzt voraus, dass alle Objekte (d.h. Client und Server) in Java programmiert sind
	+ im Unterschied z.B. zu CORBA
	+ Vorteil: nahtlose Integration in die Sprache
	+ entfernte Objekte sind (fast!) genauso wie lokale zu verwenden
	+ incl. verteilter Garbage Collection
* Integration von Objekten in anderen Programmiersprachen:
	+ ”Einwickeln“ in Java-Code über Java Native Interface (JNI)
	+ Nutzung von RMI/IIOP: Interoperabilität mit CORBA
	+ direkte Kommunikation von RMI- und CORBA-Objekten

**Beschreiben Sie, eventuell auch grafisch, die Architektur von RMI**



* Stub / Skeleton - Schicht
	+ Stub: lokales Proxy-Objekt für das entfernte Objekt
	+ Skeleton: nimmt Aufrufe entgegen und leitet sie an das richtige Objekt weiter
	+ Stub- und Skeleton-Klassen werden automatisch aus Schnittstellendefinition (Java Interface) generiert
* Ab JDK 1.2: Skeleton-Klasse wird dynamisch erzeugt
	+ RMI nutzt Reflection-Mechanismus von Java, um Verbindung zum Server-Objekt herzustellen
	+ Reflection erlaubt Abfrage der Methodendefinitionen einer Klasse und generischen Methodenaufruf zur Laufzeit
* Ab JDK 1.5: auch Stub-Klassen werden dynamisch erzeugt generische Java-Klasse Proxy

**Beschreiben Sie Infrastruktur von RMI**

**Transportschicht**

* Verbindungen zwischen JVMs
* Basis: TCP/IP
* Eigenes Protokoll: Java Remote Method Protocol (JRMP)
	+ erlaubt Tunneln der Verbindung über HTTP (wg. Firewalls)
	+ erlaubt Definition eigener Socket-Factory, z.B. zur Verwendung von Transport Layer Security (TLS bzw. SSL)
* Ab JKD 1.3 auch RMI-IIOP
	+ benutzt IIOP (Internet Inter-ORB Protocol) von CORBA
	+ damit: direkte Interoperabilität mit CORBA-Objekten

**Namensdienst: RMI Registry (Broker/Directory)**

* registriert Remote-Referenzen auf RMI-Objekte unter frei wählbaren eindeutigen Namen
* ein Client kann sich dann zu einem Namen die zugehörige Referenz holen
* technisch: Registry sendet serialisiertes Proxy-Objekt (Client-Stub) an den Client
* ggf. wird auch der Ort der Class-Datei mit übertragen (siehe später)
* RMI kann auch mit anderen Namensdiensten verwendet werden, z.B. JNDI (Java Naming and Directory Interface)

# JUnit

**Warum sollte man JUnit benutzen?**

* Standard
* Die Verwaltung der eigene Routinen muss man programmieren
* Jeder kennt die Oberfläche

**Wofür steht FIRST?**

* **Fast**: Schnell in der Ausführung, Beachtung das Tests oft durch verschiedene Arten anderer Tests durchlaufen werden muss
* **Isolated**: Tests sollten von einander unabhängig sein, kein Test sollte als Voraussetzung haben, das ein anderer Test z. B. erst laufen soll
* **Repeatable**: Wiederholbarkeit ohne das man ein zwischen Prozess erst durchführen muss oder eine Ausgangssituation erst herstellen muss
* **Self-Validating**: Test kann selbst eindeutig bestimmen, ob er erfolgreich war oder fehlgeschlagen ist
* **Timely**: aktuell, Tests sollten nicht Aufgrund von Implementierungsänderungen Ihre Status verlieren

**Welche Annotationen implementiert das Softwarepacket JUnit?**

* **Before**: kennzeichnet eine Methode, die vor jeder Testmethode
aufgerufen wird
* **After**: kennzeichnet eine Methode, die nach jeder Testmethode
aufgerufen wird
* **BeforeClass**: kennzeichnet eine Methode, die einmal aufgerufen wird
bevor die Testmethoden der Klasse aufgerufen werden
* **AfterClass**: kennzeichnet eine Methode, die einmal aufgerufen wird,
nachdem alle Testmethoden der Klasse abgearbeitet wurden
* **Ignore**: kennzeichnet eine Methode, die bei der Testausführung
ignoriert werden soll
* Alle JUnit-Testmethoden geben keine Resultate zurück und haben keine Argumente, d.h. sie sind von der Form void methode().

**Unterschied before/After und BeforeClass/AfterClass**

* Before/After beziehen sich auf die test-Methoden.
	+ Hier kann man zum Beispiele immer auf einen definierten Stand bringen
* BeforeClass/AfterClass beziehen sich auf die Testklasse.
	+ Hier kann man zum Beispiel eine DB erstellen/löschen
	+ Hier kann man zum Beispiel eine DB-Verbindung erstellen

**Wofür braucht man before/After?**

* Before/After beziehen sich auf die test-Methoden.
	+ Hier kann man zum Beispiele immer auf einen definierten Stand bringen



public class SampleClass {

 public String initializeData(){

 return "Initialize";

 }

 public String processDate(){

 return "Process";

 }

 }

public class SampleTest {

 private SampleClass sampleClass;

 @BeforeClass

 public static void beforeClassFunction(){

 System.out.println("Before Class");

 }

 @Before

 public void beforeFunction(){

 sampleClass=new SampleClass();

 System.out.println("Before Function");

 }

 @After

 public void afterFunction(){

 System.out.println("After Function");

 }

 @AfterClass

 public static void afterClassFunction(){

 System.out.println("After Class");

 }

 @Test

 public void initializeTest(){

 Assert.assertEquals("Initailization check", "Initialize", sampleClass.initializeData() );

 }

 @Test

 public void processTest(){

 Assert.assertEquals("Process check", "Process", sampleClass.processDate() );

 }

}

### public class Example {

 @BeforeClass public **static** void onlyOnce() {

 ...

 }

 @Test public void one() {

 ...

 }

 @Test public void two() {

 ...

 }

 }

### public class Example {

 DatabaseConnection database;

 @BeforeClass public static void login() {

 database= ...;

 }

 @Test public void something() {

 ...

 }

 @Test public void somethingElse() {

 ...

 }

 @AfterClass public static void logout() {

 database.logout();

 }

 }

**Welche assert gibt es?**

* assertFalse(boolean expected)
	+ Bedingung ist falsch. Wenn gleich  Fehler
* assertTrue(boolean expected)
	+ Bedingung ist wahr. Wenn ungleich  Fehler
* assertEquals(int expected, int actual)
* assertEquals(double expected, double actual, double delta)
	+ Die zwei Parameter haben den gleichen Wert (Klasse)
* assertNotNull
	+ Wenn Parameter vorhanden  okay
* assertNull
	+ Wenn Parameter ist null  okay
* assertSame
	+ gleiche Objekte, benötigt equals

**Was benötigt „assertSame“?**

* equals-Methode

**Welche Faustregel gibt es für gute Unit-Tests? AAAD**

* Arrange: Initialisierung des Tests, Definition der Startsituation.
	+ Setzen Mocks (Platzhalter für die Modultests)
	+ Before,beforeClass
* Act: die eigentliche Aktion, die das später zu verifizierende Ergebnis erzeugt.
	+ i.d.R. hier wird die Methode des Objekts aufgerufen welches getestet werden soll.
	+ Aufruf der Testklasse
* Assert: Überprüfung des zu erwartenden Ergebnis welches durch Act erzeugt wurde
* DeArrange
	+ After, afterClass

# Threads/Semaphore

**Welche Voraussetzungen müssen gegeben sein, um Threads zu benutzen?**

* Deklaration
	+ extends Threads (keine Oberklasse vorhanden)
	+ implements Runnable (Oberklasse ist vorhanden)
* implementieren
	+ public void run

**Warum sollte man Threads benutzen?**

* Wenn man umfangreiche Berechnung durchführen will

**Was benötigt man, wenn man mehrere Threads gestartet hat, die eine gemeinsame Berechnung durchführen?**

* join mit try/catch Block

**Was benötigt man, wenn mehrere gestartete Threads auf eine gemeinsame „Variable“ zugreifen sollen?**

* Mutex
* Semaphore
* wait, notify

**Motivation – Warum Threads verwenden?**

* feingranulare Parallelität der Verarbeitung
* Erzeugung/Vernichtung von Threads geht viel schneller als Prozesse
* effektive Ausnutzung mehrerer CPUs/Kerne
* Umschaltung zwischen Threads eines Prozesse ist schneller als bei Prozessen
* Dekomposition (Aufspalten) in Threads liefert Performancegewinn, insbesondere wenn Ein-/Ausgabe-beschränkte Funktionalität

**Einsatzgebiete von Threads?**

* zur Vermeidung von Blockaden der Benutzerschnittstelle;
dann kann eine Anwendung weiterhin auf Benutzereingaben reagieren
* um nach Ablauf einer bestimmten Zeit ein Ereignis auszulösen.
* Beispiel: eine Bilderfolge in einem Applet, die im Sekundentakt verändert wird
* zur Parallelisierung einer Anwendung auf mehrere Prozessoren;
* dann kann in Mehrprozessorsystemen für jeden Prozessor ein Thread gestartet werden; der parallel verarbeitbare Programmteil wird in einen Thread verpackt
* zur Bedienung von blockierenden Schnittstellen.
* Beispiel: Warten auf Netzwerkverbindungen
* Beispiele:
	+ .NET WPF – Anwendungen: Erzeugung der Anwendung in 2 Threads, einen für die UI und einen für die normalen Berechnung
	+ Android – Java: 2 Threads für UI und normales Handling

**Unterschied Prozess und Threads?**

* Prozess ist schwergewichtig
* Keine gemeinsamen Variablen zwischen Prozessen
* Gemeinsamen Variablen zwischen Threads

**Beschreiben Sie den Lebenszyklus eines Threads?**

* created
	+ new Thread()
	+ mit start() geht es zu alive
* alive/running
* non- running
* close





**Warum ist stop() in einem Thread depricated?**

* Der Thread wird **sofort** gestoppt.
* Dateien etc. könnten noch offen sein
* Besser interrupted() bzw. isinterrupted()

**Warum sollte man join(long millicsec) nicht benutzen?**

* Der Thread wartet auf die anderen.
* Wenn die anderen Threads noch Zeit brauchen, kann er trotzdem weiterlaufen!
	+ Fehler

**Was ist race-Condition?**

* Mehrere Threads bemühen sich um ein Betriebsmittel.
* Es werden keine mutexe oder Semaphore verwendet, so dass zwei Threads eine Variable „zeitgleich“ ändern können. Die erste Änderung wird dann überschrieben.
* Abhilfe:
	+ Synchronized, Mutex, Semaphore, wait und notify

**Was ist besser als stop() mit in einem Thread?**

* Besser interrupted()
* In der While-Schleife wird dieses Flag abgefragt

**Was ist ein Deadlocks?**

* Deadlocks sind eine Tücke im Multithreading
* Mindestens zwei Threads wartet gegenseitig auf ein Betriebsmittel
	+ Thread A hat Tablet
		- Thread A wartet auf Stift
	+ Thread B hat Stift
		- Thread B wartet auf Tablet

**Voraussetzungen für Deadlocks?**

Bedingungen für das Entstehen von Deadlocks

* Wechselseitiger Ausschluss
	+ Ein Betriebsmittel ist entweder verfügbar oder komplett von einem anderen Thread belegt.
* Belegungs- und Wartebedingung
	+ Ein Thread belegt ein Betriebsmittel und wartet **zusätzlich** auf die Zuteilung weiterer Mittel.
* Un-Unterbrechbarkeit
	+ Belegte Betriebsmittel können einem Thread nicht entzogen werden, sondern müssen vom Thread selbst freigegeben werden.
* Zyklische Wartebedingung
	+ Es gibt eine zyklische Kette aus zwei oder mehr Threads, die Betriebsmittel nutzen, welche vom nächsten Thread benötigt werden

**Abhilfe für Deadlocks?**

* Statt mehrerer Betriebsmittels ein Gesamtpaket Betriebsmittels vergeben.
	+ Alles oder nichts
* Ein Masterthread killt ein oder mehrere Threads

# JDBC

**Definition von JDBC**

* Java ist eine Interpreter-Sprache
* direkter Zugriff auf eine native-Datenbank ist schwierig.
	+ Informatik, PostgreSQL, Firebird
* Es gibt aber auch Java-Datenbanken
	+ HSQLDB
	+ Apache Derby

**Welche Grundlagen braucht man für JDBC?**

* JDBC-Treiber (\*.jar)
* Connection (login)
* Schleife:
	+ Statement
	+ ResultSet
	+ ResultSetMetaData
	+ While-schleife
		- Ausgabe der Ergebnisse einer Abfrage

**Nennen Sie Anwendungsgebiete für JDBC**

* Java ist eine Interpretersprache.
* Es kann nur über JNI auf native Datenbanken zugreifen.
	+ JNI = Java Native Interface
* Alternativ:
	+ HSQLDB
	+ Apache Derby

**Motivation von Transaktionen?**

* Alles oder nichts
* Eine Datenbanktransaktion besteht aus einer Gruppe von Teilaktionen, die in einer oder mehreren Datenbanktabellen erfolgreich ausgeführt werden müssen, bevor sie endgültig festgeschrieben (**Commit**) werden können.
* Schlägt eine der Teiltransaktionen in der Gruppe fehl, werden alle anderen Teilaktionen ebenfalls rückgängig gemacht (**Rollback**).
* Transaktionen verhindern, dass die Datenbank in einen **inkonsistenten Zustand** gerät, wenn ein Problem bei der Durchführung einer der Aktionen auftritt, aus denen sich die Transaktion

**Warum gibt es Prepared Statements?**

* Java-Interface PreparedStatement
* PreparedStatement sind SQL-Anweisungen, die Platzhalter haben.
* Diese Platzhalter werden dann jeweils dem Statement übergeben.
	+ PreparedStatement ps = conn.prepareStatement(
		- "SELECT Wohnort, Strasse FROM Angestellte WHERE Vorname=**?** AND Nachname=**?**");
	+ ps.setString(1, "Willi");
	+ ps.setString(2, "Meier");
	+ ResultSet rs = ps.executeQuery();
* Weitere Alternative:
	+ PreparedStatement auf dem Server
		- mittels PLSQL (ProgrammLanguage SQL)
		- Übertragen werden nun nur noch die Parameter

# XML

**Welche Vorteile hat die Verwendung von XML/XSD gegenüber DTD?**

* XML-Format, also Standard
* Min/MaxOccurs flexibler (1,?,+,\*)
* Echte Datentypen
* min/maxInclusive (logischer Wertebereich, negative Preise)
* Prüfungsdatei ist auch im XML-Format
* Weitere XML-Software:
	+ Export nach HTML, Excel etc.
* Mehrere Namensräume
* Ableitungen
* Choice

**Vorteile XML**

* „ XML leicht zu verarbeiten
* „ Schnittstellen für die meisten Programmiersprachen
* „ Umfang von Fehlerüberprüfungen und Ausnahmebehandlungen begrenzt durch Wohlgeformtheitsregeln

**Nachteile**

* Sehr speicherintensiv (keine Komprimierung)
* Ungeeignet zur Speicherung von Binärdaten (MIME)
* Langsamer Zugriff (rel. DB)
* Kaum Optimierung für Zugriffsgeschwindigkeit (ID, IDREF zum schnelleren Finden von Knoten)
* Bei wenig Speicherplatz/Bandbreite nicht geeignet für große Datenmengen

**Wie kann man eine Klasse in XML implementieren?**

* ComplexType

**Wie kann man Attribute/Eigenschaften genauer spezifizieren?**

* SimpleType

**Unterschied XML und XHTML?**

* XHTML hat festdefinierte Tag’s.
* XML hat keine festdefinierte Tag’s.

**Unterschied wellformed zu valid?**

* Prüfung mit Wellformed ist einfacher
* Valid beinhaltet Wellformed

**Man sollte man eine Eigenschaft als Attribute nehmen, wann nicht?**

* Attribut kann nur **einmal** vorkommen
* Hat man mehrere „Instanzen“ sollte man eine SimpleType nehmen