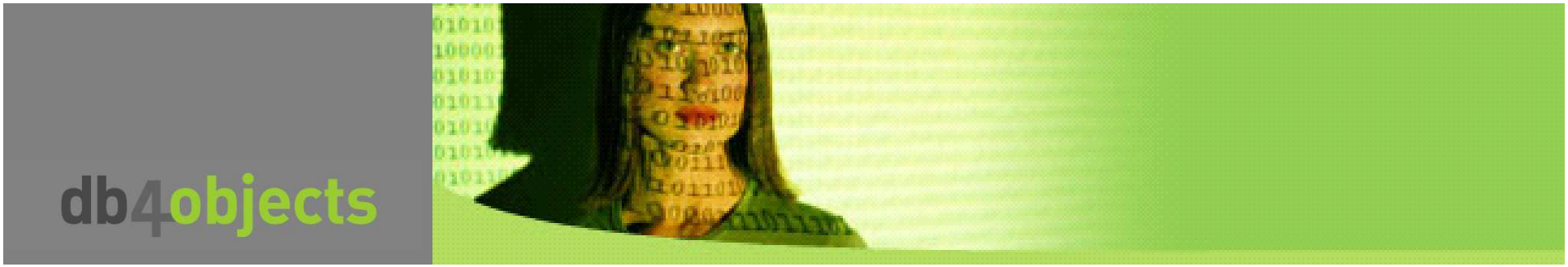


Native Queries
Datenbankabfragen in reinem Java

Carl Rosenberger
Chief Software Architect, db4objects Inc.
Java Forum Stuttgart 2006, 6. Juli 2006

Herkömmliche Datenbankabfragen sind nicht objektorientiert

- SQL, EJBQL, JDOQL basieren auf Zeichenketten (Strings)
- keine Überprüfung durch die Entwicklungsumgebung
- keine automatische Refakturierung
- Abfragen durchbrechen objektorientierte Kapselung
- ständiger Wechsel zwischen Implementationssprache und Abfragesprache
- schlechte Unterstützung für den Bau wiederverwendbarer Komponenten

Abfragen in SQL, EJBQL, JDOQL

```
// SQL
String sql = "SELECT * FROM Student student WHERE student.age < 20";
ResultSet rs = statement.executeQuery(sql);

// EJBQL
String sql =
  "select student from Student as student where student.age < 20";
List students = entityManager.createQuery(sql);

// JDOQL
String where = "age < 20";
persistenceManager.newQuery(Student.class, where);
```

Ziel

- Abfragen sollen in reinem Java ausgedrückt werden können

- 100% nativ
- 100% objektorientiert
- 100% typsicher
- 100% durch den Compiler überprüfbar
- 100% refakturierbar

Ausdruck

- einfach nur Java

```
student.getAge() < 20
```

Rückgabewert

- true => Bestandteil der Ergebnismenge

```
{
  return student.getAge() < 20;
}
```

Methode

- einfach nur Java

```
public boolean match(Student student){
  return student.getAge() < 20;
}
```

Klasse

- einfach nur Java

```
new Predicate(){
  public boolean match(Student student){
    return student.getAge() < 20;
  }
}
```

Generics

- erlaubt festen Kontrakt einer #match() Methode

```
new Predicate <Student>(){
  public boolean match(Student student){
    return student.getAge() < 20;
  }
}
```

Die komplette Abfrage

- einfach nur Java

```
List <Student> students = database.query <Student>(
  new Predicate <Student>(){
    public boolean match(Student student){
      return student.getAge() < 20;
    }
  }
);
```

Tipparbeit ?

- => Eclipse code template

```
List <${extent}> students = database.query <${extent}>(
  new Predicate <${extent}>(){
    public boolean match(${extent} candidate){
      return true;
    }
  }
);
```

Komplexeres Beispiel

```
final String grade = "A";
final String countryName = "Germany";

List <Student> scholarshipCandidates = database.query <Student>(
  new Predicate <Student>(){
    public boolean match(Student student){
      return (( student.getAge() < 20
          &&    student.getGrade().equals(grade) )
          ||    student.getCountry().getName().equals(countryName))
          && !  student.hasScholarship();
    }
  }
);
```

Der Welt einfachste Native Query Engine

- im Speicher
- gegen Collections
- komplett mit folgendem Code

```
public <T> List<T> filter (List<T> list, Predicate<T> predicate){
  List<T> result = new ArrayList<T>();
  for (T candidate : list){
    if(predicate.match(candidate)){
      result.add(candidate);
    }
  }
  return result;
}
```

Optimierung

```
List <Student> students = database.query <Student>(
  new Predicate <Student>(){
    public boolean match(Student student){
      return student.getAge() < 20;
    }});

// => db4o SODA syntax

Query query = database.query();
query.constrain(Student.class);
query.descend("age").constrain(new Integer(20));
List students = query.execute();
```

Möglichkeiten der Optimierung

- Vor dem Kompilieren
  - Quellcode
- Nach dem Kompilieren
  - Bytecode Instrumentierung
- Während des Ladens
  - ClassLoader
- zur Laufzeit
  - Bytecode Analyse

```
List <Student> students =
  database.query <Student>(
    new Predicate <Student>(){
      public boolean match(Student s){
        return s.getAge() < 20;
      }
    }
  );
```

- Open Source Implementation in db4o
  - db4onqopt Projekt in den db4o Sourcen

Nicht Optimierbar

- GUI
- Benutzerinteraktion
- Thread Erzeugung
- Veränderung und Speicherung persistenter Objekte

Optimierbar

- Variablendeklaration
- Statische Methodenaufrufe
- Java !
- => alles was nicht ausdrücklich nicht erlaubt ist

Laufzeitverhalten

- Eingeschränkter Speicher
- Timeouts

Fehlerbehandlung

- Rückgabewert true => zur Ergebnismenge
- Alle anderen Fälle => nicht zur Ergebnismenge

- Callbacks und Logging für Fehler zur Laufzeit und bei der Optimierung

Exkurs C#

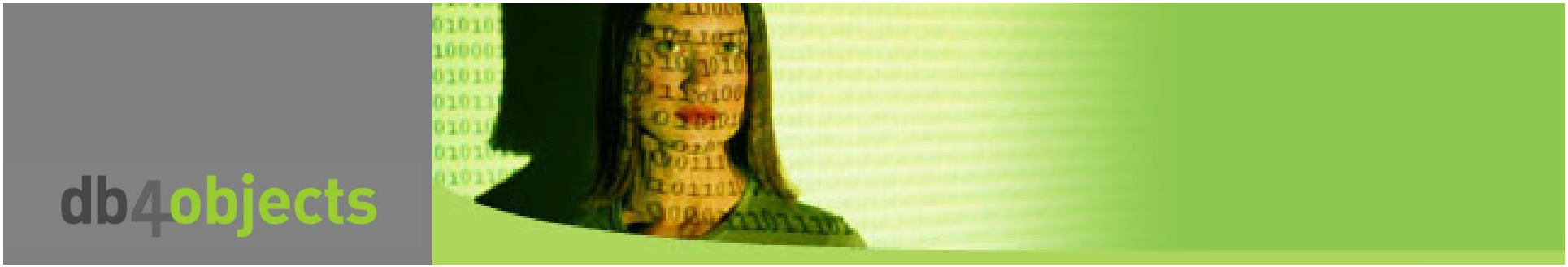
anonyme Delegates erlauben noch elegantere Ausdrucksform

```
IList <Student> students = db.Query <Student> (
  delegate(Student student){
    return student.Age < 20;
  });
```

Vorteile von Native Queries

- 100% nativ
- 100% objektorientiert
- 100% typsicher
- 100% durch den Compiler überprüfbar
- 100% refakturierbar
- Autoergänzung durch die Entwicklungsumgebung
- Aufbau komplexer Abfrageframeworks möglich
- sehr gute Wiederverwendbarkeit
- gegen Collections ausführbar

- optimiert => sehr schnell



Vielen Dank!

- Download **Whitepaper zu Native Queries**
- http://www.db4o.com/about/productinformation/whitepapers/

- Download db4o 5.x
- http://download.db4o.com/