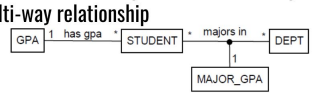
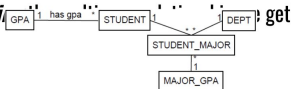
UML:A **foreign key** is a field of a table that refers to the primary key of another table. **Cardinality** denotes how many entities participate in the relationship.
One-Many The * next to STUDENT indicates record in DEPT can be related to zero or more STUDENT records. The 1 next to DEPT indicates that each STUDENT record must have exactly one major department. **Relational Schema 2.** If table T1 contains foreign key to table T2, then create relationship between classes T1 and T2. Annotation next to T2 is "1" Annotation next to T1 is "0..1" if foreign key also a key; otherwise, annotation is "*". **3.** Add attributes to each class. Attributes should include all fields of its table, except for foreign keys and artificial key fields. (PID, FK)_1, (FK,...)_0. **Natural Keys:** If the attribute is also unique, it can replace the key. **Inadequate Relationships**This happened because of we oversimplified requirement spec interpretation.**Use Reification To Remove Many-Many.**

Reify Multi-Way Relationships

- We can combine ***majors in*** and ***has major gpa*** into multi-way relationship



- ***Reifyi***

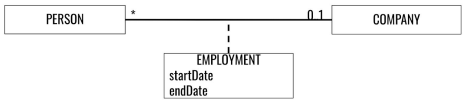


get

Aggregation is a part of/ contains. describes a part-whole or part-of relationship. No lifecycle dependency. (0..1). **Composition** describes an owns a relationship. Strong lifecycle dependency. (1..1) **Generalization** describes an is a/ is a kind of, include, relationship. (1..1). **Attribute:**have a. **Association:** verb relation. **A functional dependency** X1,X2,…Xn → Y asserts that two records with same values Xi, must also have same value of Y. FD is super key. Prof teaches exactly one course. FD: Prof → CourseId.

Association Classes

- It's about the relationship between the two
- Use association class EMPLOYEMENT



Normal Form: A table is in first normal form if it contains no repeating fields / columns(Data must depend on the key)

First Normal Form Example (3/3)

- Remove redundant row

CID	FirstName	Surname	Telephone
123	Robert	Ingram	555-861-2025
456	**Jane**	**Wright**	**555-403-1659**
457	**Jane**	**Wright**	**555-776-4100**
789	Maria	Fernandez	555-808-9633

- Revised ***Customer*** table and new ***Telephones*** table

CID	FirstName	Surname
123	Robert	Ingram
456	**Jane**	**Wright**
789	Maria	Fernandez

CID	Telephone
123	555-861-2025
456	**555-403-1659**
456	**555-776-4100**
789	555-808-9633

2NF – The Whole Key

- Second normal form refactors tables that have columns that do not **depend on whole *primary key***
- Consider the following table

CID	Semester	#Seats	CourseName
CS1500	2009-1	100	Programming
CS1500	2009-2	100	Programming
CS5200	2009-1	200	Databases
CS5200	2010-1	150	Databases
CS5610	2009-2	120	Web Design

- **CourseName** column does not **depend on whole** PK, only on one of the keys: CID

3NF – Nothing But the Key

- Every non-key attribute must provide a fact about ***the key, the whole key, and nothing but the key***

Course	Year	Prof	Prof Date of Birth
Web Dev	2012	Alice Wonder	26 November 1865
Intro to DB	2012	Bob Marley	6 February 1945
C++	2011	Charly Garcia	23 October 1951
Web Dev	2013	Bob Marley	6 February 1945
Intro to DB	2013	Alice Wonder	26 November 1865

- Removing birth date from table:

Course	Year	Prof
Web Dev	2012	Alice Wonder
Intro to DB	2012	Bob Marley
C++	2011	Charly Garcia
Web Dev	2013	Bob Marley
Intro to DB	2013	Alice Wonder

- And moving it to a dedicated table

Prof	Prof Date of Birth
Alice Wonder	26 November 1865
Bob Marley	6 February 1945
Charly Garcia	23 October 1951

Select – returns same columns as input table, removes some rows. **Project** – returns same rows as input table, removes some columns. **Sort** – returns same input table, rows in a different order. **Rename** – returns same table, one column with a different name. **Extend** – returns same table, additional column with computed value. **Groupby** – returns table, one row for each group of input records. **Product** – returns table containing all possible combinations of records from input tables. **Join** – connects tables together. Equivalent to selection of a product. **Semijoin** – returns table with records from input table1 that match some record in input table2. **Antijoin** – returns table with records from input table1 that do not match records in input table2. **Union** – returns table with records from both tables. **Outer Join** – returns table with records of join, and with non-matching records padded with nulls. SORT(table, [first, last]), RENAME(Q6, oldname, newname). **GROUP BY:** Extend combines fields in the same record horizontally. GroupBy combines fields several records vertically
Records are grouped if fields have the same values. Fields from grouped records are aggregated. **groupby**(TABLE, {group, fields}, {aggegate, funcs}). Q12 = **groupby**(STUDENT, {MajorId}, {Min(GradYear), Max(GradYear)}) **"Number of students having a major"**: groupby(STUDENT, {}, {Count(MajId)}). **join**(T1, T2, P) ≡ **select**(product(T1, T2), P). **Semijoin** retrieves records that HAVE a match. **Anitjoin** retrieves records that DON'T HAVE a match Exactly the opposite.

Example: Regular Join Vs Semijoin

- Consider tables ***Employee*** and ***Dept***

Employee

Name	EmpId	DeptName
Harry	3415	Finance
Sally	2241	Sales
George	3401	Finance
Harriet	2202	Production

Dept

DeptName	Manager
Sales	Bob
Sales	Thomas
Production	Katie
Production	Mark

- The ***Join*** on the ***DeptName***:

Name	EmpId	DeptName	Manager
Sally	2241	Sales	Bob
Sally	2241	Sales	Thomas
Harriet	2202	Production	Katie
Harriet	2202	Production	Mark

- The ***Semijoin*** on the ***DeptName***:

Name	EmpId	DeptName
Sally	2241	Sales
Harriet	2202	Production

3 inheritance strategies
1. Joined AKA Normalized
2. Single Table AKA Denormalized
3. Table per Class (Hybrid)

Group By Example…

- Consider the following ***ORDER*** table

OId	OrderDate	OrderPrice	***Customer***
1	2008/11/12	1000	**Alice**
2	2008/10/23	1600	*Charlie*
3	2008/09/02	700	**Alice**
4	2008/09/03	300	**Alice**
5	2008/08/30	2000	Bob
6	2008/10/04	100	*Charlie*

…Group By Example

- We can calculate the sum for each customer

```
SELECT Customer, SUM(OrderPrice)
FROM Orders
GROUP BY Customer
```

Customer	SUM(OrderPrice)
Alice	2000
Charlie	1700
Bob	2000

Then Grouping Would Yield

```
Q83 = select e.SectionId, count(e.EId) as NumAs
      from ENROLL e
      where e.Grade = 'A'
      group by e.SectionId
```

SectionId	NumAs
1	2
2	3
3	1

```
Q84 = select max(q.NumAs)
      from Q83 q
```

MaxAs
3

- But now how do we relate that max # of A's to the ID? How about ***adding*** the ***sectionId*** to the select?

```
Q84a = select max(q.NumAs) as MaxAs,
       q.SectionId
       from Q83 q
```

MaxAs	SectionId
3	2

?

```
Q84b = select q.SectionId
       from Q83 q
       where q.NumAs = max(q.NumAs)
```

```
Q85 = select Q83.SectionId
      from Q83, Q84
      where Q83.NumAs = Q84.MaxAs
```

```
SELECT column_name,
aggregate_function(column_name)
FROM table_name
WHERE column_name operator value
GROUP BY column_name
HAVING aggregate_function(column_name)
operator value
```

- HAVING clause allows ***WHERE*** to use ***aggregate*** functions

Consider Orders table

Oid	*OrderDate*	*OrderPrice*	*Customer*
1	2008/11/12	1000	Boone
2	2008/10/23	1600	Nilsen
3	2008/09/02	700	Boone
4	2008/09/03	300	Boone
5	2008/08/30	2000	Jensen
6	2008/10/04	100	Nilsen

```
SELECT Customer, SUM(OrderPrice) FROM Orders
GROUP BY Customer
HAVING SUM(OrderPrice) < 2000
```

Customer	*SUM(OrderPrice)*
Nilsen	1700

INSERT INTO table (name..)
VALUES (values..)
JDBC:DAO: Data Access Objects

Create a Singleton MovieDao

```
package com.jga.movie.dao;
public class MovieDao {
   private static MovieDao instance = null;
   private MovieDao() {}
   public static MovieDao getInstance() {
      if(instance == null)
         instance = new MovieDao();
      return instance;
   }
}
```

Declare SQL and method

```
   String findAllMoviesSql =
                        "SELECT * FROM movies";
   public List<Movie> findAllMovies() {
      List<Movie> movies =
        new ArrayList<Movie>();
      Connection connection = null;
      Statement statement = null;
      ResultSet results = null;
      // next set of slides
   }
```

Execute an SQL Query

```
   try {
      Class.forName("com.mysql.jdbc.Driver");
      connection = DriverManager.getConnection(
         "jdbc:mysql://URL/DATABASE",
         "jannunzi", "pa$$w0rd");
      statement = connection.createStatement();
      results = statement
         .executeQuery(findAllMoviesSql);
      // next set of slides
while(results.next()) {
   String imdbId = results.getString("imdbId");
   String title = results.getString("title");
   String plot = results.getString("plot");
   String poster = results.getString("poster");
   Movie movie =
      new Movie(title, imdbId, plot, poster);
   movies.add(movie);
}
   } catch (ClassNotFoundException e) {...
   } catch (SQLException e) {...
   } finally {
      try {
         connection.close();
      } catch (SQLException e) {
         e.printStackTrace();
      }
   }
   return movies;
```

Build SQL Query String

```
private static String CREATE_MOVIE =
   "INSERT INTO movies (imdbId, title, plot,
   poster) VALUES (?, ?, ?, ?)";
private static String USERNAME = "jannunzi";
private static String PASSWORD = "pa$$w0rd";
private static String CONN_STRING =
"jdbc:mysql:///DB";

public void createMovie(Movie movie) {

try {
   Class.forName("com.mysql.jdbc.Driver");
   Connection connection =
      DriverManager.getConnection(
         "jdbc:mysql://URL/DB",
         "jannunzi", "pa$$w0rd");
   PreparedStatement statement =
   connection.prepareStatement(CREATE_MOVIE);
   // next slide
} catch (ClassNotFoundException e) { ... }
```

```
statement.setString(1, movie.getImdbId());
statement.setString(2, movie.getTitle());
statement.setString(3, movie.getPlot());
statement.setString(4, movie.getPoster());
statement.executeUpdate();
```

JPA:

@Entity: table stored in the database. @Id: primary key @Table: table name @column: column names in a table

```
@Entity
@Table(name="STUDENT")
public class Student {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    @Column(name="STUDENT_NAME", length=50, nullab
    private String name;

    // other fields, getters and setters
}
```

```
import javax.persistence.*;
@Entity
public class User {
  @Id
  @GeneratedValue
    (strategy=GenerationType.IDENTITY)
  private int id;
}
public List<User> findUserByUsername
    (@Param("username") String username);

@Query("SELECT user FROM User user
        WHERE user.username=:username")
public List<User> findUserByUsername
    (@Param("username") String username);
```

JPA maps classes to tables, prc fields, and instances

Use repository's **findById()** to retrieve one record

```
@GetMapping("/api/users/{userId}")
public User findUserById
  (@PathVariable("userId") Integer id) {
 return userRepository.findById(id).get();
}
```

Try **http://localhost:8080/api/users/1**

Map data model to equivalent REST API endpoi

```
@RestController
public class UserService {
  @GetMapping("/api/users")
  public List<User> findAllUsers() { }
}
```

Retrieve all records

Use repository's **findAll()** method to retrieve all records

```
@Autowired
UserRepository userRepository;
@GetMapping("/api/users")
public List<User> findAllUsers() {
 return (List<User>) userRepository.findAll()
}
```

DELETE FROM table WHERE..
[UPDATE table SET column1=value, column2=value2,... WHERE...]

Retrieve users by username

```
@GetMapping("/api/users")
public List<User> findAllUsers(@RequestParam
  (name="username", required=false) String uname,
{ if(username != null) {
    return userRepository.
      findUserByUsername(uname); }
  return (List<User>) userRepository.findAll();
}
```

Posting New Data to RESTful Services

```
@PostMapping("/api/users")
public User createUser
  (@RequestBody User user) {
    return userRepository
      .save(user);
```

Updating Data in RESTful Services

```
@PutMapping("/api/users/{userId}")
public User updateUser(
  @PathVariable("userId") int id,
  @RequestBody User newUser) {
User user = userRepository.findById(id).get();
user.set(newUser);
return userRepository.save(user);}
```

Deleting Data from RESTful Servi

```
@DeleteMapping("/api/users/{userId}")
public void deleteUser
(@PathVariable("userId") int id) {
  userRepository.deleteById(id);
}
```

One to Many Relations

```
public class Faculty extends User {
  ...
  @OneToMany(mappedBy="author")
  private List<Course> authoredCourses;
}

  @Entity
  @Table(name="courses")
  public class Course { ...
    @Id ...
    private int id;
    private String name;
    @ManyToOne()
    @JsonIgnore
    private Faculty author;
  }
```

Generated JPA Course Schem

```
CREATE TABLE courses (
 id int(11) NOT NULL AUTO_INCREMENT,
 name varchar(255) DEFAULT NULL,
 author_id int(11) DEFAULT NULL,
 PRIMARY KEY (id),
 KEY ... (author_id),
 FOREIGN KEY (author_id) REFERENCES user (id)
)
```

Implementing `authoredCourse()`

```
public class Faculty extends User { ...
  private List<Course> authoredCourses;
  public void authoredCourse(Course course)
{   this.authoredCourses.add(course);
    if(course.getAuthor() != this) {
      course.setAuthor(this);
}}}
```

Implement Post New C

```
@RestController
public class CourseService {
  @Autowired
  CourseRepository courseRepository;
  @PostMapping("/api/courses")
  public Course createCourse
    (@RequestBody Course course) {
    return courseRepository.save(course);
}}
```

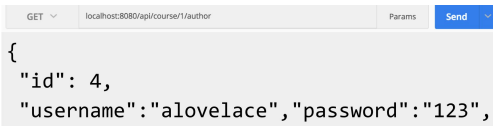
Many to Many `FacultyService.authoredCourse()`

```
@PutMapping("/api/faculty/{fId}/authored/{cId}")
public void authoredCourse(
      @PathVariable("fId") int fId,
      @PathVariable("cId") int cId) {
   Faculty faculty = facultyRepository.findOne(fId
   Course course   = courseRepository.findOne(cId)
   course.setAuthor(faculty);          ⇐
   courseRepository.save(course);      use either
   faculty.authoredCourse(course);     ⇐
   facultyRepository.save(faculty);
}
```

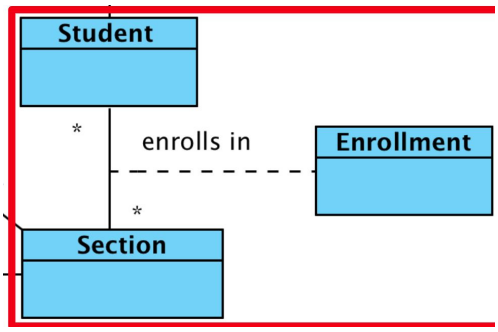
Implement Find Authored

```
public class FacultyService {...
  @GetMapping
  ("/api/faculty/{facultyId}/authored")
  public Iterable<Course> findAuthoredCourses
       @PathVariable("facultyId") int fId) {
     Faculty faculty =
       facultyRepository.findOne(fId);
     return faculty.getAuthoredCourses();
}}
```

```
public class CourseService {
  @GetMapping("/api/course/{courseId}/author")
  public Faculty findCourseAuthor(
       @PathVariable("courseId") int cId) {
     Course course =
       courseRepository.findOne(cId);
     return course.getAuthor();
  }
}
```




```
{
 "id": 4,
 "username":"alovelace","password":"123",
 "firstName":"Ada","lastName":"Lovelace",
 "office": "321B", "tenure": true
}
```




```
@Entity
public class Section { ...
  @Id ...
  private String name;
  @ManyToMany(mappedBy="enrolledSections")
  @JsonIgnore
  private List<Student> enrolledStudents;
}
```

Add Bi-Directional Relation

```
public class Student extends User { ...
 @ManyToMany
 @JoinTable(name="ENROLLMENT",
   joinColumns=@JoinColumn(name="STUDENT_ID"
     referencedColumnName="ID"),
   inverseJoinColumns=@JoinColumn(name=
     "SECTION_ID",
   referencedColumnName="ID"))
 @JsonIgnore
 private List<Section> enrolledSections;
```

ENROLLMENT

```
CREATE TABLE enrollment (
  student_id int(11) NOT NULL,
  section_id int(11) NOT NULL,
  KEY ... (section_id),
  KEY ... (student_id),
  FOREIGN KEY (section_id)
            REFERENCES section (id),
  FOREIGN KEY (student_id)
            REFERENCES user (id)
)
```

POST NEW Section:

```
package com.jga.repositories;
import
  org.springframework.data.repository.*;

import com.jga.models.Section;
public interface SectionRepository extends
  CrudRepository<Section, Integer> { }
```

```
@RestController
public class SectionService {
  @Autowired
  SectionRepository sectionRepository;
  @PostMapping("/api/section")
  public Section createSection
       (@RequestBody Section section) {
    return sectionRepository.save(section);
}}
```

```
{
  "name": "SECTION01"
}
```

Body | Cookies | Headers (3) | Test Re

Pretty | Raw | Preview | JSON

```
{
    "id": 1,
    "name": "SECTION01",
    "enrolledStudents": null
}
```

Implement

```
public class SectionService {
  @GetMapping("/api/section")
  public Iterable<Section>
    findAllSections() {
      return
        sectionRepository.findAll()
}}
```

http://localhost:8080/api/sectio

```
[ { "id": 1, "name": "SECTION01",
    "enrolledStudents": []
  },
  { "id": 2, "name": "SECTION02",
    "enrolledStudents": []
}]
```

Add Section.enrollStudent()

```
public class Section {
   private List<Student> enrolledStudents;
   public void enrollStudent(Student student) {
      this.enrolledStudents.add(student);
      if(!student.getEnrolledSections()
         .contains(this)) {
            student.getEnrolledSections()
               .add(this);
}}}
```

SectionService.enrollStudentInSection(

```
public class SectionService {
   @Autowired
   StudentRepository studentRepository;
   @PostMapping("/api/section/{zId}/student/{sId}")
   public void enrollStudentInSection(
         @PathVariable("zId") int zId,
         @PathVariable("sId") int sId) {
      Section section = sectionRepository.findOne(zId);
      Student student = studentRepository.findOne(sId);
      section.enrollStudent(student);
      sectionRepository.save(section);
}}
```

Retrieve Section Enrollments

```
public class SectionService {
   @GetMapping("/api/section/{sId}/student")
   public Iterable<Student>
      findSectionEnrolledStudents(
            @PathVariable("sId") int sId) {
         Section section =
               sectionRepository.findOne(sId);
         return section.getEnrolledStudents();
}}
```

Joined JPA

```
@Entity
@Table(name = "JOINED_BASE_QUESTION")
@Inheritance(strategy=InheritanceType.JOINED)
public class BaseQuestionJoined {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY
  private int id;
  private int points;
  private String title, description, instructions; ...
}
```

TrueOrFalseQuestionJoined.java

```
@Entity
@Table(name =
   "JOINED_TRUE_OR_FALSE_QUESTION")
public class TrueOrFalseQuestionJoined
  extends BaseQuestionJoined {
  @Column(name = "IS_TRUE", nullable = false)
  private Boolean isTrue;
}
```

RepositoryJoined.java

```
public interface BaseQuestionRepositoryJoined
  extends CrudRepository
    <BaseQuestionJoined, Integer>{}
public interface TrueOrFalseQuestionRepositoryJoined
  extends CrudRepository
    <TrueOrFalseQuestionJoined, Integer>{}
public interface FillInTheBlankQuestionRepositoryJoined
  extends CrudRepository
    <FillInTheBlankQuestionJoined, Integer>{}
```

/api/inheritance/joined/fill

```
public void testCreateFillQuestion() {
  FillInTheBlankQuestionJoined q = new
     FillInTheBlankQuestionJoined();
  q.setDescription("descriptions 234");
  q.setInstructions("instructions 234");
  q.setPoints(234);
  q.setTitle("title 234");
  q.setVariables("variables 234");
  fillRepo.save(q);
}
```

Single Table

```
@Entity
@Table(name = "SINGLE_BASE_QUESTION")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public class BaseQuestionSingle {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private int id;
  private int points;
  private String title, description, instructions; ...
}
```

TrueOrFalseQuestionSingle.java

```
@Entity
@Table(name =
   "SINGLE_TRUE_OR_FALSE_QUESTION")
public class TrueOrFalseQuestionSingle
  extends BaseQuestionSingle {
  @Column(name = "IS_TRUE")
  private Boolean isTrue;
}
```

HTTP GET requests is the simplest and most common type of HTTP Requests

HTTP GET is what browsers use to retrieve documents from HTTP servers

Embed parameters in URL as **path** or **query** parameters

http://find/user/by/id/123?minsalary=10000

```
package com.example.myapp.services;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController
@RestController
public class HelloWorldService {
  @GetMapping("/hello")
  public String sayHello() {
    return "Hello World";
}}
```

This simple example allow remote process to access a simple string by invoking a URL path ending in /hello

Web service can be parameterized with variables in the pa

```
@GetMapping("/hello/{name}")
public String sayHelloToName
    (@PathVariable("name") String nameVariable) {
  return "Hello " + nameVariable;
}
```

Responding:

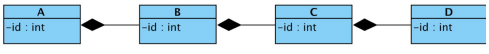
```
@GetMapping("/message")
public Message getMessage() {
  return new Message();
}
```

HTTP POST requests allow embedding request data in HTTP body in addition, or instead of, path and request parameters. Moreover, passing sensitive information in HTTP GET requests as path or request parameters is not secure. HTTP BODY in HTTP POST requests can be encrypted for secure client/server data communication

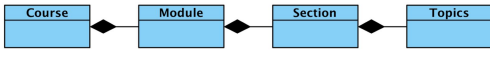
Use **@RequestBody** to retrieve HTTP

```
@PostMapping("/api/user")
public User createUser
    (@RequestBody User newUser) {
  return newUser;
}
```

Representational state transfer
POST: Create new instances
GET: Read existing instances
PUT: Update existing instances
DELETE: Delete existing instances



POST	/A	Create new entity of type A. Data in HTTP body
GET	/A	Retrieve all instances of entity type A, e.g., array
GET	/A/123	Retrieve single instance, whose ID is 123
PUT	/A/234	Update instance, whose ID is 234. Data in body
DELETE	/A/345	Delete existing instance, whose ID is 345



POST	/modules/123/sections	Create new section in module 123
GET	/modules/234/sections	Retrieve all sections in module 234
GET	/sections/345	Retrieve section 345
PUT	/sections/456	Update section 456
DELETE	/sections/567	Delete section 567