Lecture 11

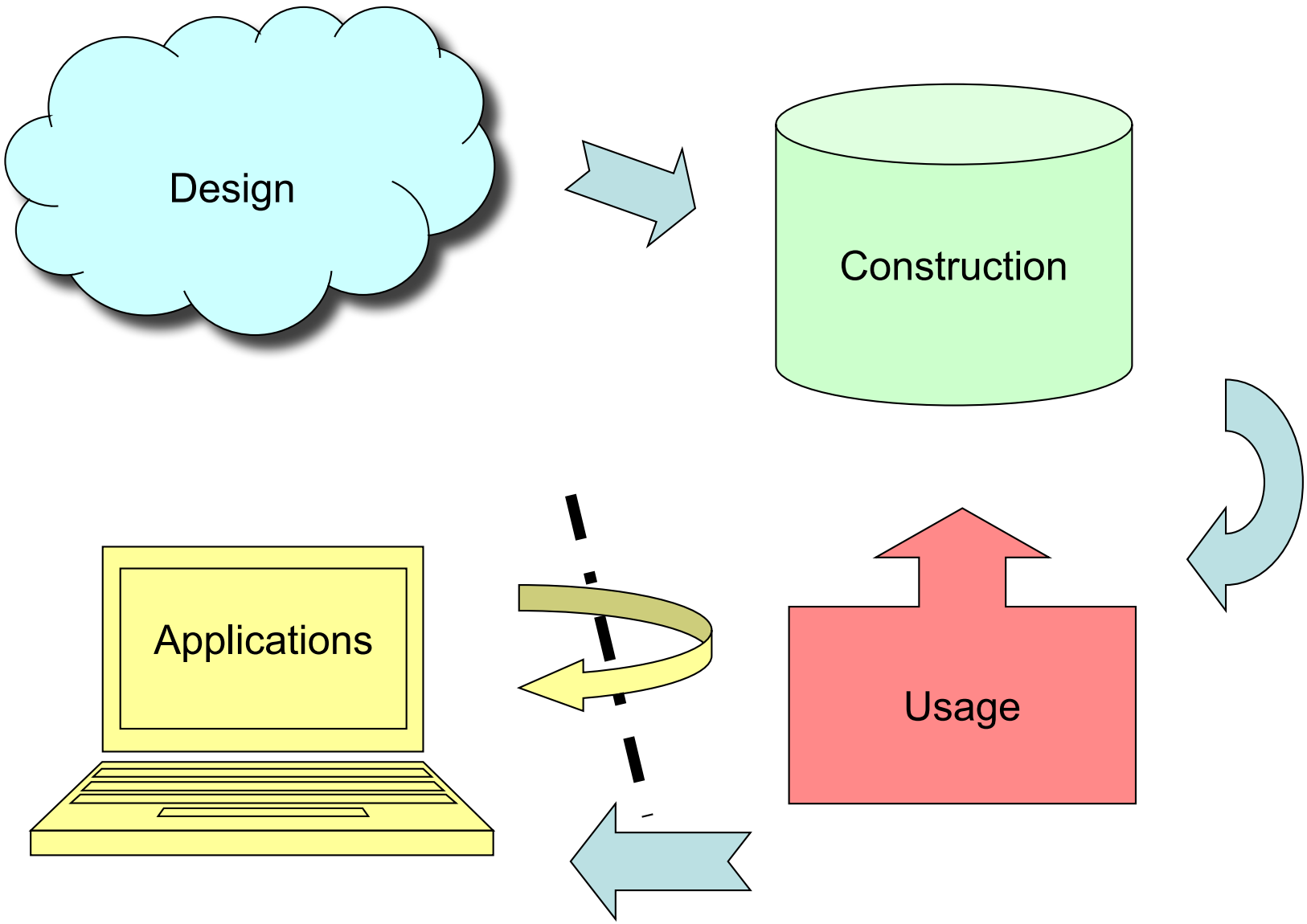
Database Applications

JDBC

SQL Injection

Authorization

Course Objectives



JDBC

- JDBC = Java DataBase Connectivity
- JDBC is Java's *call-level interface* to SQL DBMS's.
  - A library with operations that give full access to relational databases, including:
    - Creating, dropping or altering tables, views, etc.
    - Modifying data in tables
    - Querying tables for information
    - …

JDBC Objects

- JDBC is a library that provides a set of classes and methods for the user:
  - DriverManager
    - Handles connections to different DBMS. Implementation specific.
  - Connection
    - Represents a connection to a specific database.
  - Statement, PreparedStatement
    - Represents an SQL statement or query.
  - ResultSet
    - Manages the result of an SQL query.

Getting connected

```
private static final String HOST =
  ”ate.ita.chalmers.se”;
private static final String DB = ”exampledb”;
private static final String USER = username;
private static final String PWD = password;

Class.forName("org.postgresql.Driver");

Properties props = new Properties();
props.setProperty("user",USERNAME);
props.setProperty("password",PASSWORD);


Connection myCon =
  DriverManager.getConnection(”jdbc:postgresql://” +
    HOST + ”/” + DB, props);
```

- Will also be done for you on the lab, except username and password.

Statements

- A **`Statement`** object represents an SQL statement or query, including schema-altering statements.
- A **`Statement`** object represents one statement at a time, but may be reused.

```
Statement myStmt = myCon.createStatement();
```

A statement is associated with a particular connection

Using statements

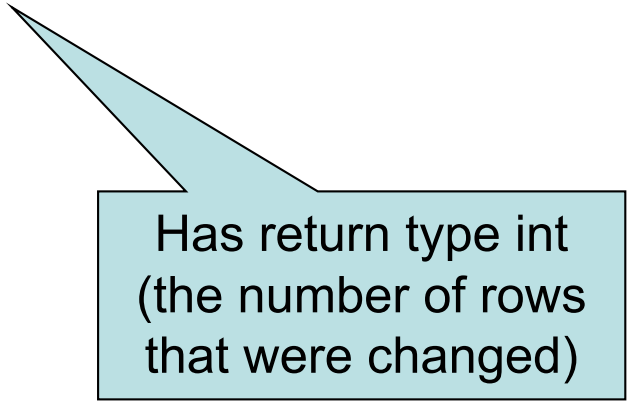
- **`Statement`** objects have two fundamental methods:
  - **`ResultSet executeQuery(String query)`**
    - Given a string, which must be a query, run that query against the database and return the resulting set of rows.
  - **`int executeUpdate(String update)`**
    - Given a string, which must be a non-query, run that update against the database.
    - Note that a JDBC update is not an SQL update, but rather an SQL modification (which could be an update).

Example:

```
String myInsertion =
  ”INSERT INTO Courses VALUES (’TDA357’, ’Databases’)”;

Statement myStmt = myCon.createStatement();

myStmt.executeUpdate(myInsertion);
```



Exceptions in JDBC

- Just about anything can go wrong!
  - Syntactic errors in SQL code.
  - Trying to run a non-query using **`executeQuery`**.
  - Permission errors.
  - …
- Catch your exceptions!

```
try {
      // database stuff goes in here
} catch (SQLException e) { … }
```

Executing queries

- The method **`executeQuery`** will run a query against the database, producing a set of rows as its result.
- A ResultSet object represents an interface to this resulting set of rows.
  - Note that the **`ResultSet`** object is not the set of rows itself – it just allows us to access the set of rows that is the result of a query on some **`Statement`** object.

ResultSet

- A ResultSet holds result of an SQL query.
  - **`boolean next()`**
    - Advances the ”cursor” to the next row in the set, returning false if no such rows exists, true otherwise.
  - **`X getX(i)`**
    - **`X`** is some type, and **`i`** is a column number (index from 1).
    - Example: **`rs.getInt(1)`**

      returns the integer value of the first column of the current row in the result set **`rs`**.

ResultSet is not a result set!

- Remember a **`ResultSet`** is more like a cursor than an actual set – it is an interface to the rows in the actual result set.
- A **`Statement`** object can have one result at a time. If the same **`Statement`** is used again for a new query, any previous **`ResultSet`** for that **`Statement`** will no longer work!

Quiz!

What will the result be?

```
Statement myStmt = myCon.createStatement();
ResultSet rs =
  myStmt.executeQuery("SELECT * FROM Courses");
while (rs.next()) {
  String code = rs.getString(1);
  String name = rs.getString(2);
  System.out.println(name + " (" + code + ")");
  ResultSet rs2 = myStmt.executeQuery(
    "SELECT teacher FROM GivenCourses " +
    "WHERE course = '" + code + "'");
  while (rs2.next())
    System.out.println(" " + rs2.getString(1));
}
```

SQLi!

Due to overuse of the same **`Statement`**, only the first course will be printed, with teachers. After the second query is executed, **`rs.next()`** will return false.

Two approaches

- If we need information from more than one table, there are two different programming patterns for doing so:
  - Joining tables in SQL
    - Join all the tables that we want the information from in a single query (like we would in SQL), get one large result set back, and use a ResultSet to iterate through this data.
  - Use nested queries in Java
    - Do a simple query on a single table, iterate through the result, and for each resulting row issue a new query to the database (like in the example on the previous page, but without the error).

Example: Joining in SQL

```
Statement myStmt = myCon.createStatement();
ResultSet rs =
  myStmt.executeQuery(
    "SELECT code, name, period, teacher " +
    "FROM Courses, GivenCourses " +
    "WHERE code = course " +
    "ORDER BY code, period");

String currentCourse, course;
while (rs.next()) {
  course = rs.getString(1);
  if (!course.equals(currentCourse))
    System.out.println(rs.getString(2));
  System.out.println("  Period " + rs.getInt(3) +
                     ": " + rs.getString(4));
  currentCourse = course;
}
```

Compare with previous row to see if this is a new course. If it is, print its name.

Example: Using nested queries in Java

```
Statement cStmt = myCon.createStatement();
Statement gcStmt = myCon.createStatement();
ResultSet courses = cStmt.executeQuery(
     "SELECT code, name " +
     "FROM Courses " +
     "ORDER BY code");

while (courses.next()) {
  String course = courses.getString(1);
  System.out.println(courses.getString(2));
  ResultSet gcourses = gcStmt.executeQuery(
     "SELECT period, teacher " +
     "FROM GivenCourses
     "WHERE course = '" + course + "' " +
     "ORDER BY period");
  while (gcourses.next()) {
    System.out.println("  Period " + gcourses.getInt(1) +
                       ": " + gcourses.getString(2));
  }
}
```

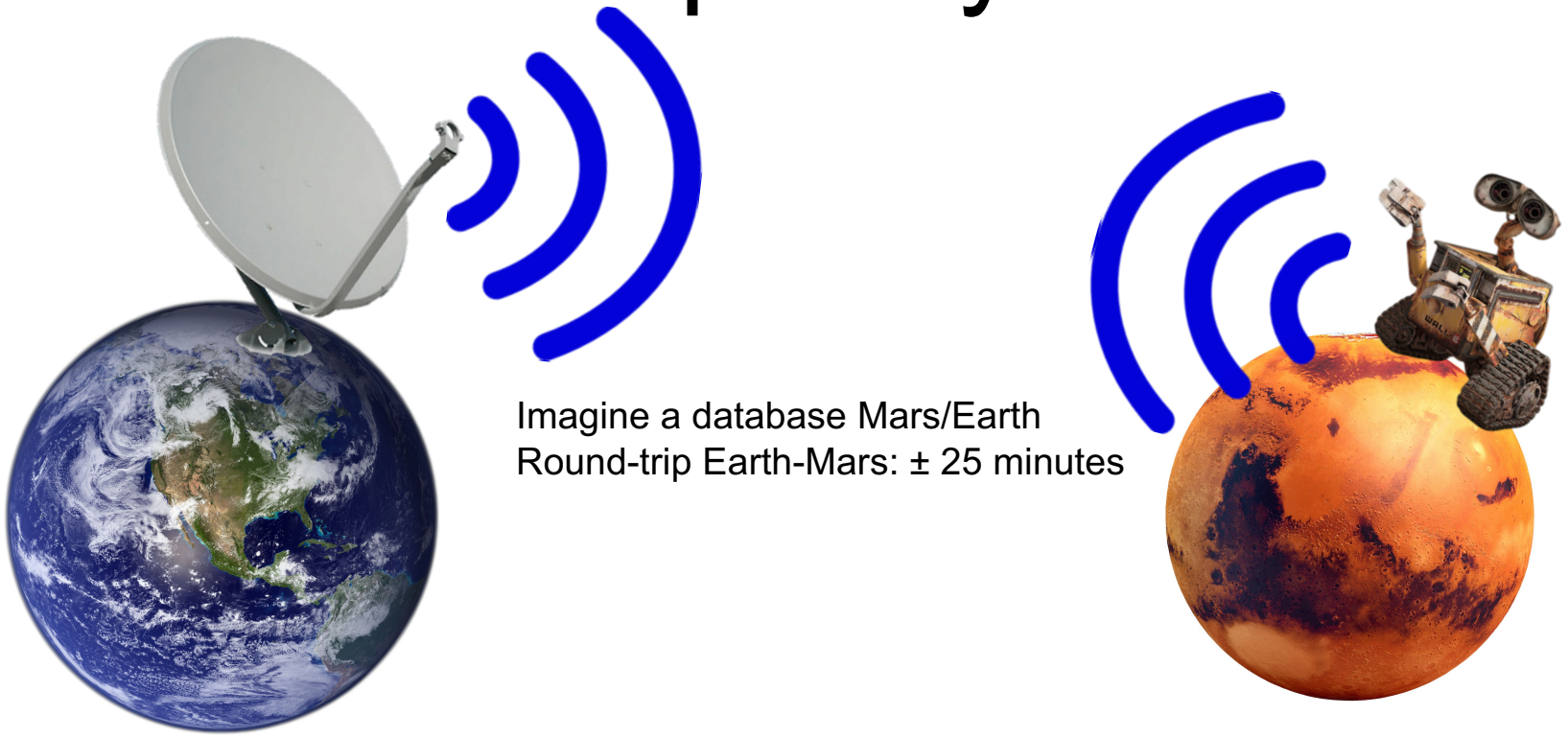
SQLi!

Find the given courses for each course separately with an inner query.

Comparison

- Joining in SQL
  - Requires only a single query.
  - Everything done in the DBMS, which is good at optimising.
- Nested queries
  - Many queries to send to the DBMS
    - communications/network overhead
    - compile and optimise many similar queries
  - Logic done in Java, which means optimisations must be done by hand.
  - Limits what can be done by the DBMS optimiser.

Push complexity to DBMS



Imagine a database Mars/Earth
Round-trip Earth-Mars: ± 25 minutes

- CPUs are fast (nanoseconds per instruction)
- Network communication is slow (milliseconds per packet)
  - Millions of times slower than a CPU computation!!
- Place your complexity on the DBMS if possible
  - Avoid costly round-trips over network!

Dynamically generated queries

```
SELECT * FROM UserInfo WHERE username = <user input>;
```

- Goal: pass user-input to DBMS as part of the query
  - E.g. asking for information on a certain user

- Good assumption: User are attackers
  - Always sanitize your inputs!
- **SQL Injection (SQLi) is the most common vulnerability on the Web today**

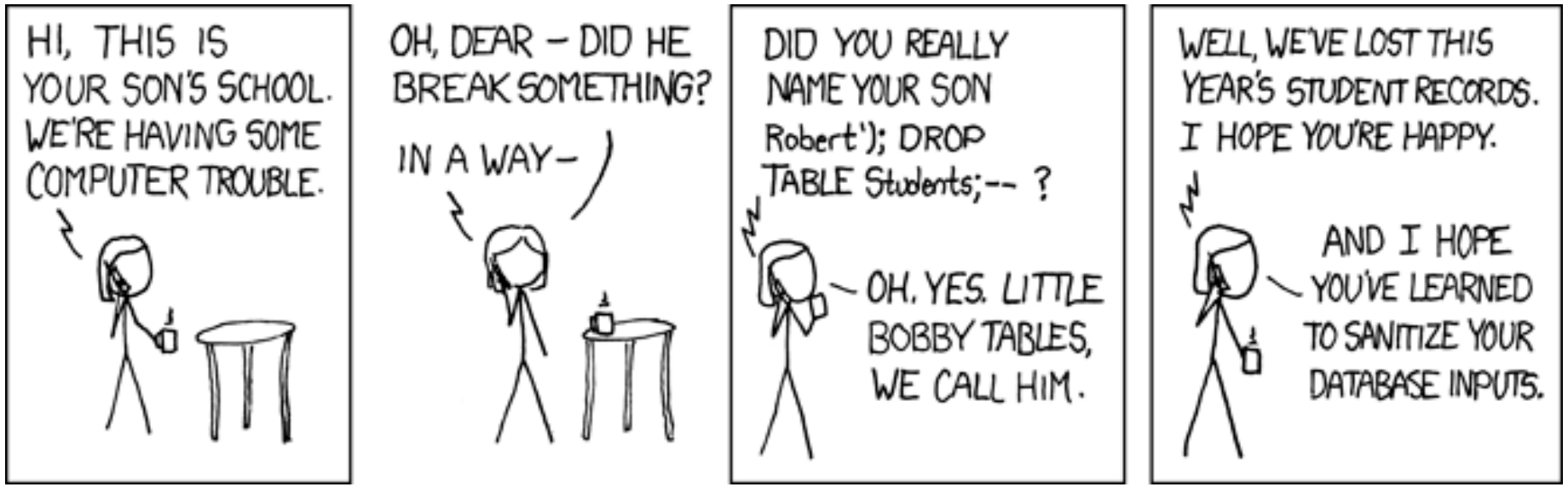
A1-Injection	Injection flaws, such as SQL, OS, and LDAP injection occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's hostile data can trick the interpreter into executing unintended commands or accessing data without proper authorization.

https://www.owasp.org/index.php/Top_10_2013-Top_10

Ethical Hacking

Never poke around with security on systems without explicit permission

(Consider that you may be dealing with critical systems such as nuclear powerplants or hospital equipment)



Dynamically generated queries: Naïve approach

```
myStmt.executeQuery("SELECT * FROM UserInfo
WHERE username = '"+ username +"'");
```

SQLi!

- String concatenation will result in SQLi

Username = **abc**

```
SELECT * FROM UserInfo WHERE
      username = 'abc'
```

Username = **x' OR '1'='1**

```
SELECT * FROM UserInfo WHERE
      username = 'x' OR '1'='1'
```

Username = **x' UNION (SELECT uid, password, 'x', 'y' FROM UserPasswords) --**

```
SELECT * FROM UserInfo WHERE username = 'x'
      UNION (SELECT uid, password, 'x', 'y'
            FROM UserPasswords) --'
```

SQL Injection: sqlmap

- SQLmap
  - "automatic SQL injection and database takeover tool"
  - http://sqlmap.org/
- **USE ONLY WITH PERMISSION!**
- Automatically discovers SQL vulnerabilities, determines best SQLi attack and extracts entire database
- **Prevent SQL Injection Vulnerabilities in your applications**
  - This tool is used in the wild, don't be a victim of it

PreparedStatement

- We can parametrize data in a statement.
  - Data that could differ is replaced with **?** in the statement text.
  - **?** parameters can be instantiated using functions **setX(int index, X value)**.

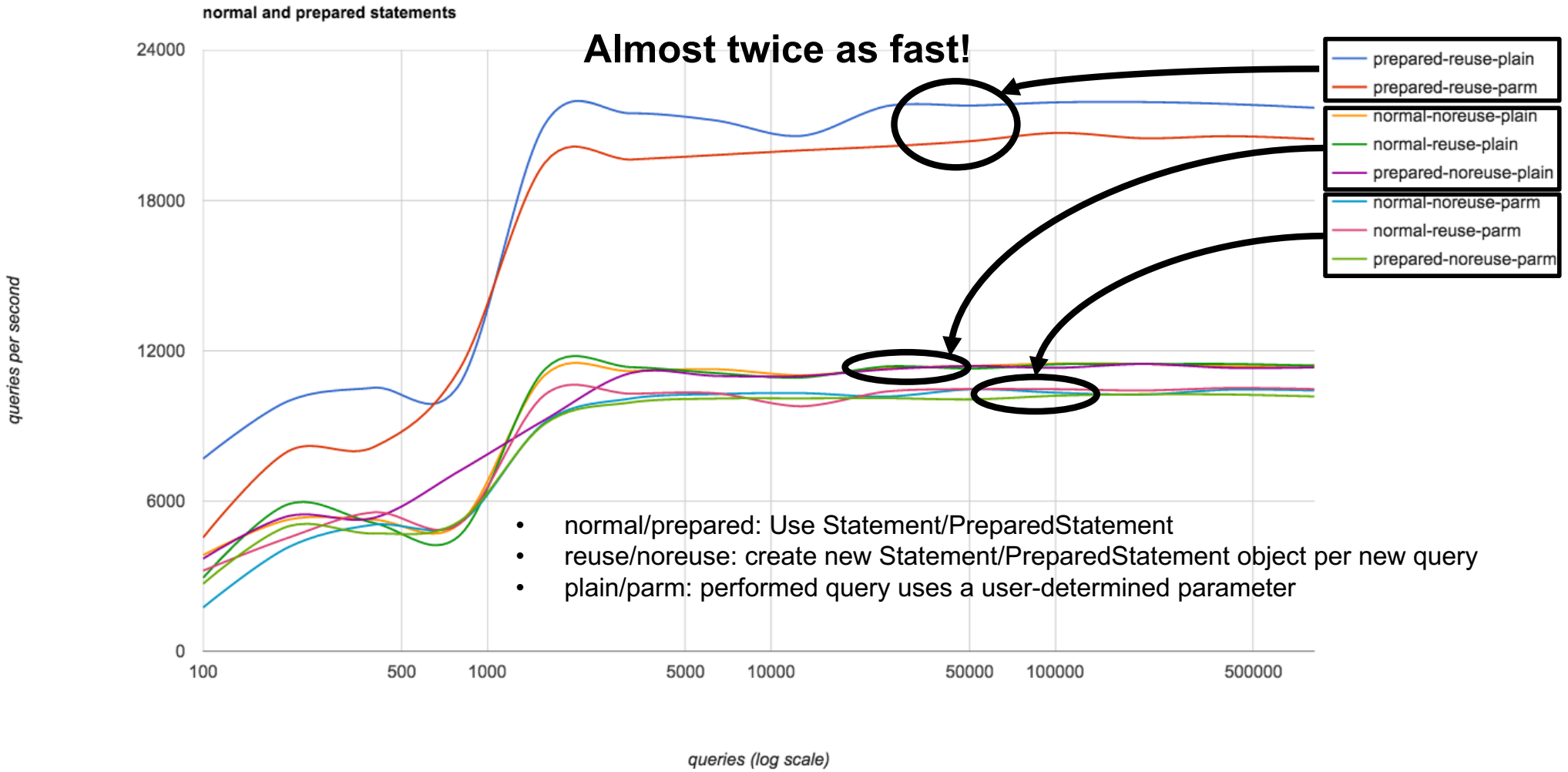
```
PreparedStatement myPstmt =
 myCon.prepareStatement(
   ”INSERT INTO Courses VALUES (?,?)”);

myPstmt.setString(1, ”TDA356”);
myPstmt.setString(2, ”Databases”);
```

PreparedStatement is superior: Reason 1 – Security

- PreparedStatements are designed to prevent SQL injections
  - The query is separated from the attacker input by using ’?’ placeholders
  - They know how to safely encode parameters are Strings, Integers and others
  - Because of this strict separation and encoding, attackers can not inject into the SQL query

PreparedStatement is superior: Reason 2 – Performance



- normal/prepared: Use Statement/PreparedStatement
- reuse/noreuse: create new Statement/PreparedStatement object per new query
- plain/parm: performed query uses a user-determined parameter

PreparedStatement is superior: Reason 3 – easier to read/write

```
myStmt.executeQuery("SELECT * FROM UserInfo
WHERE username = '"+ username +"'");
```

Missing "!!!

SQLi!

```
conn.prepareStatement("SELECT * FROM UserInfo
WHERE username = ?");
```

- Because of the placeholders, PreparedStatements are easier to both read and write
  - No messing with brackets and escaping characters

Prevent SQL injection!

Do not mix user input with your queries!!

Use PreparedStatement!!

Summary JDBC

- **`DriverManager`**
  - Register drivers, create connections.
- **`Connection`**
  - Create statements or prepared statements.
  - Close when finished.
- ~~**`Statement`**~~
  - ~~Execute queries or modifications.~~
- **`PreparedStatement`**
  - Execute a particular query or modification, possibly parametrized. Good practice for security reasons.
- **`ResultSet`**
  - Iterate through the result set of a query.

Play with SQLi

- http://redtiger.labs.overthewire.org/
  - All SQL injection challenges
- http://overthewire.org/wargames/natas/
  - All web challenges, with SQLi in later levels

Database Authorization

Authorization

- Not every user can be allowed to do everything.
  - Some data are secret and may only be seen by some users.
  - Some data are high integrity and may only be modified by certain users.

Database vs file system

- A (UNIX) file system has:
  - Privileges on files.
  - Three different privileges: read, write, execute
  - Three levels of access: owner, group, all
- A database has:
  - Privileges on schema elements (tables, views, triggers, etc.)
  - Nine different privileges.
  - Any number of levels of access – each user can be given different access.

Privileges on relations

- **`SELECT (`*`attributes`*`) ON `*`table`***
  - Allows the user to select data from the specified table.
  - Can be parametrized on attributes, meaning the user may only see certain attributes of the table.
- **`INSERT (`*`attributes`*`) ON `*`table`***
  - Allows the user to insert tuples into the table.
  - Can be parametrized on attributes, meaning the user may only supply values for certain attributes of the table. Other attributes are then set to NULL.

Privileges on relations

- **`DELETE ON` *`table`***
  - Allows the user to delete tuples from the table.
  - Cannot be parametrized on attributes.
- **`UPDATE (`*`attributes`*`) ON` *`table`***
  - Allows the user to update data in the table.
  - Parametrizing means the user may only update values of certain attributes.

Other privileges

- **`REFERENCES (`*`attributes`*`) ON `*`table`**
  - Allows the user to create a foreign reference to (attributes of) that table.
- **`TRIGGER ON `*`table`**
  - Allows the user to create triggers for events on that table.
- **`EXECUTE ON `*`procedure`**
  - Allows the user to execute the procedure or function, and use it in declarations.
- **`USAGE, UNDER, TRUNCATE, CREATE, ALL,`**

  ...

Quiz!

What privileges are needed to perform the following insertion?

```
INSERT INTO Lectures(course, period, weekday)
 SELECT course, period, 'Monday'
 FROM   GivenCourses G
 WHERE  NOT EXISTS
          (SELECT course, period
           FROM   Lectures L
           WHERE  L.course = G.course
                  AND L.period = G.period
                  AND weekday = 'Monday');
```

We need privileges **INSERT** on **Lectures(course, period, weekday)**, **SELECT** on **GivenCourses(course, period)**, and **SELECT** on **Lectures(course, period, weekday)**.

EXECUTE and TRIGGER

- When writing a trigger, the body may perform selections and modifications.
  - The user who writes the trigger must have all the necessary privileges to perform those operations, plus the **`TRIGGER`** privilege.
  - The user that sets off the trigger needs only the privilege to perform the triggering event (e.g. an insertion). Everything that happens in the trigger is considered done by its creator.
- The same thing goes for procedures and functions – it is the privileges of the creator that decides what operations may be performed, and the user needs only **`EXECUTE`**.

Granting privileges

- You have all possible privileges on elements that you have created.
- You may grant privileges to other users on those elements.
  - A user is referred to by an *authorization ID*, which is typically a user name.
  - There is a special authorization ID, *public*
  - Granting a privilege to *public* makes it available to all users.

GRANT statement

- Granting a privilege in SQL:

```
GRANT list of privileges
ON element
TO list of authorization Ids;
```

  – Example:

```
GRANT SELECT(course, period, teacher)
ON    GivenCourses
TO    public;
```

WITH GRANT OPTION

- A user that can grant privileges on some element can choose to grant **`WITH GRANT OPTION`**.
  - The grantee can then grant this privilege further.
  - Example:

```
GRANT SELECT(course, period, teacher)
ON    GivenCourses
TO    nibro WITH GRANT OPTION;
```

Revoking privileges

- Privileges can be revoked with the inverse statement:

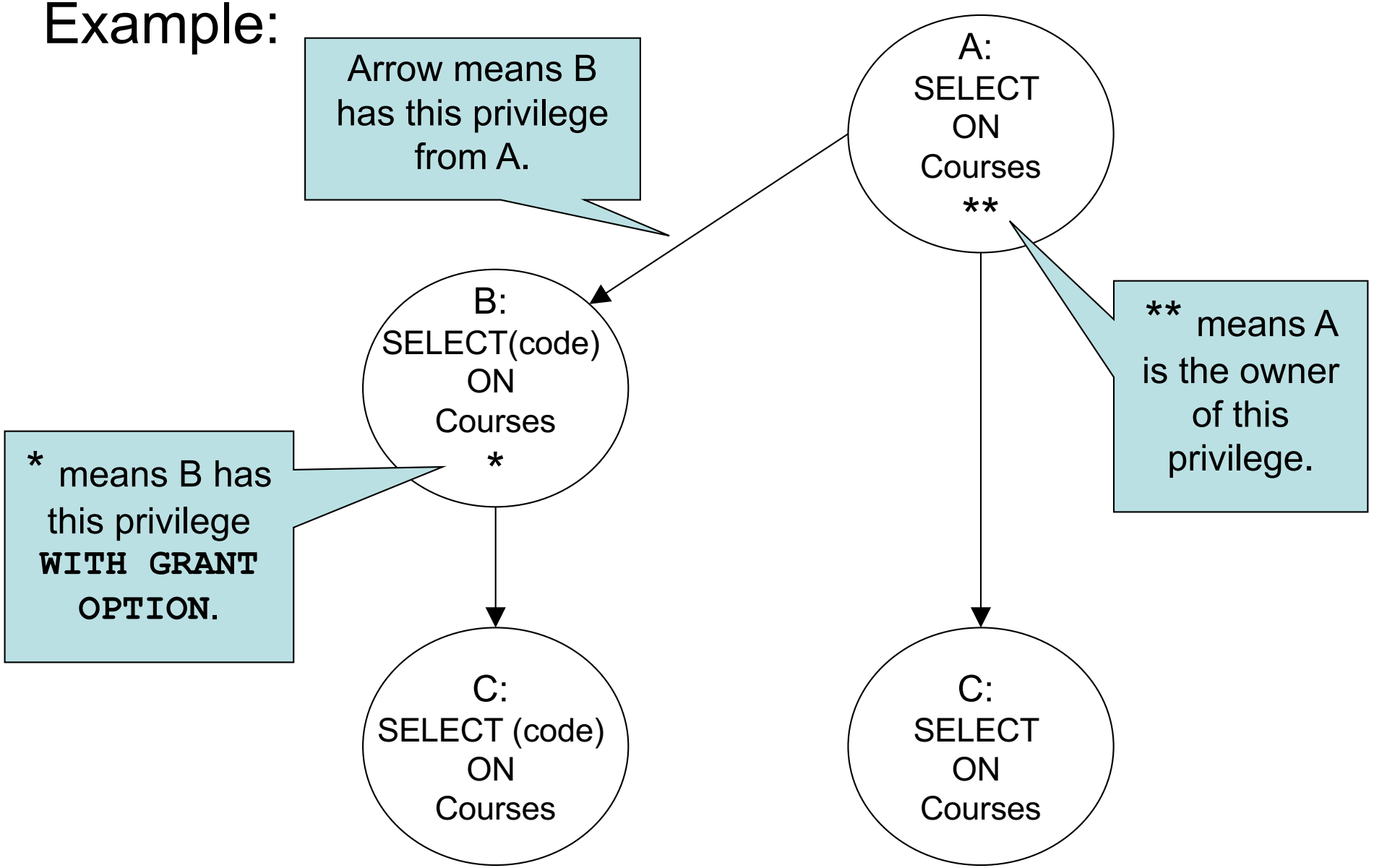
```
REVOKE list of privileges
ON     element
FROM   list of authorization Ids;
```

- Your grant of these privileges can no longer be used by these users to justify their use of the privilege.
  - But they may still have the privilege because they have it from another independent source.
- **`CASCADE`** and **`RESTRICT:`** like **`UPDATE/DELETE`** policies (see foreign keys from before)

Grant diagrams

- Nodes = user + privilege + option
  - Option is either owner, **`WITH GRANT OPTION`**, or neither.
  - **`UPDATE ON T`**, **`UPDATE(a) ON T`**, **`UPDATE(b) ON T`** and **`UPDATE ON T WITH GRANT OPTION`** all live in different nodes.
- Edge X → Y means that node X was used to grant Y.

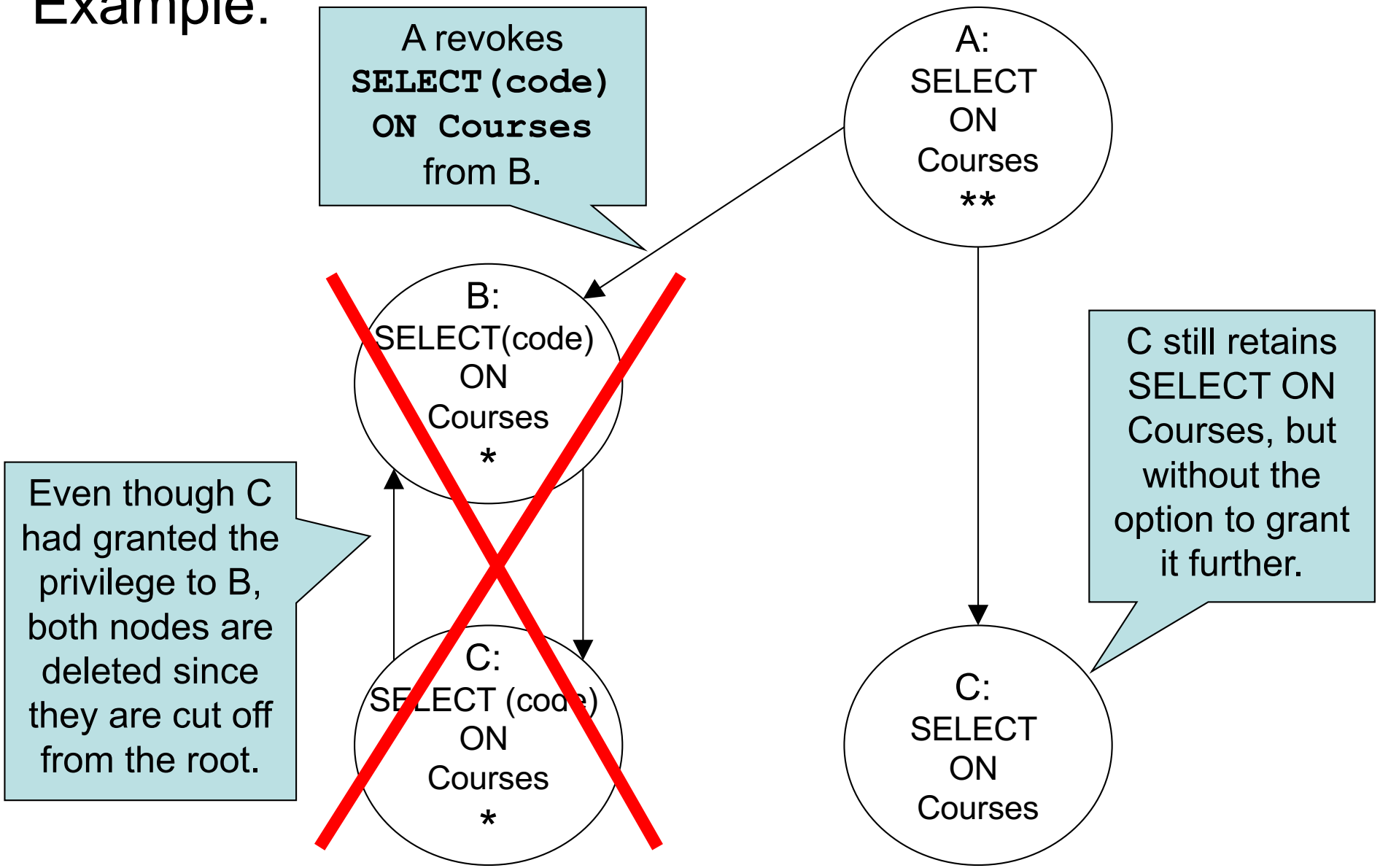
Example:


Arrow means B has this privilege from A.
A: SELECT ON Courses **
B: SELECT(code) ON Courses *
** means A is the owner of this privilege.
* means B has this privilege WITH GRANT OPTION.
C: SELECT (code) ON Courses
C: SELECT ON Courses

Manipulating edges

- If A grants P to B, we draw an edge from AP* (or AP**) to BP(* if with grant option).
- Revoking a privilege means deleting the edge corresponding to the privilege.
- Fundamental rule: User U has privilege P as long as there is a path from XP** to either UP, UP* or UP**, where X is the owner of P.
  - Note that X could be U, in which case the path is 0 steps.

Example:


A revokes
SELECT(code)
ON Courses
from B.
A:
SELECT
ON
Courses
**
B:
SELECT(code)
ON
Courses
*
C:
SELECT (code)
ON
Courses
*
C:
SELECT
ON
Courses
Even though C had granted the privilege to B, both nodes are deleted since they are cut off from the root.
C still retains SELECT ON Courses, but without the option to grant it further.

Summary Authorization

- Privileges in SQL
  - **`SELECT, INSERT, DELETE, UPDATE, REFERENCE, TRIGGER, EXECUTE`** ...
- Granting and revoking privileges
  - Authentication IDs, public
  - **`WITH GRANT OPTION`**
- Grant diagrams

Next time, Lecture 12

Transactions