Software Tools
Data Access Layers

Part II - Lecture 3

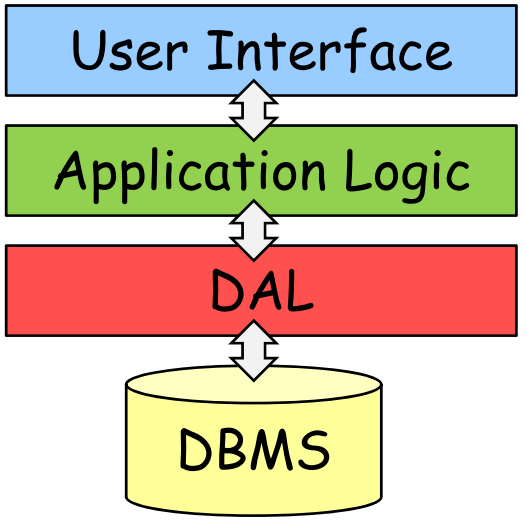
Today's Outline

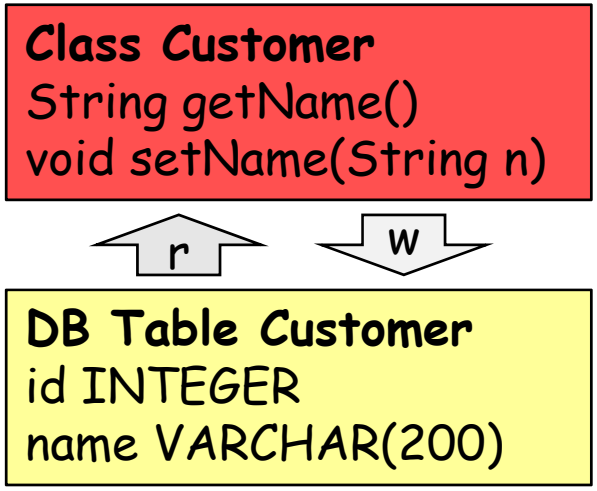
- Data Access Layers
- PDStore
- Assignment 2 Project

Data Access Layers

Data Access Layer (DAL)

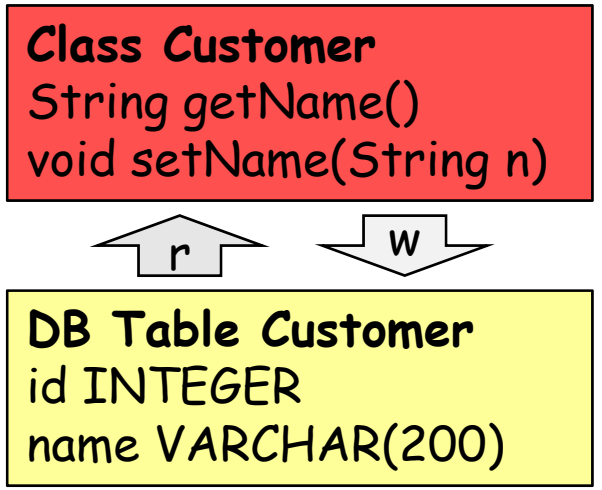
- Application layer that provides functionality for convenient DB access
- Enables the use of OO classes to read and write from/to the DB (instead of having to use SQL)
- Advantages:
  - Higher level of abstraction
  - Separation of concerns makes application easier to maintain
  - Application is independent of particular DB management system (DBMS)





Developing DALs: Writing DALs Manually

- Use a DB access API such as JDBC
- Low level: need to deal with SQL and possibly with DB specific code
- Tedious: writing SQL for getters/setters is very repetitive
- Maintenance problem when data model specification changes (DAL needs to be changed as well)

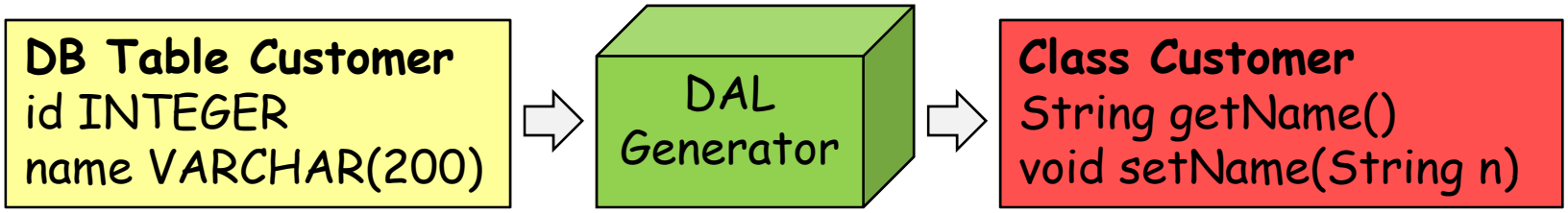



```
String getName() throws SQLException {
  Statement s = connection.createStatement();
  ResultSet r = s.executeQuery(
    "SELECT name FROM Customer WHERE id=" + id);
  String name = null;
  if (r.next())  name = r.getString(1);
  s.close();  return name;
}
```

Developing DALs: Generating DALs

Generating the DAL from data model specification

- For each data type, a class with getters and setters is generated by a DAL generator
- Getters/setters read from and write to the DB
- Generator may support different DBMS
- When the data model specification changes, simply re-run the generator to get an updated DAL
- Example: PDStore

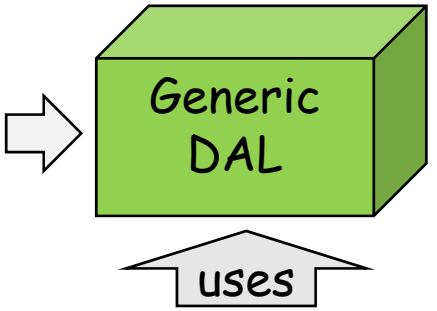


Developing DALs: Using a Generic DAL

Using a generic DAL with a mapping specification

- Use your own classes for the persistent data types
- Specify how the classes should be mapped to the DB
- Use generic functions to begin/commit transactions and load/save objects from and to the DB
- Example: Hibernate

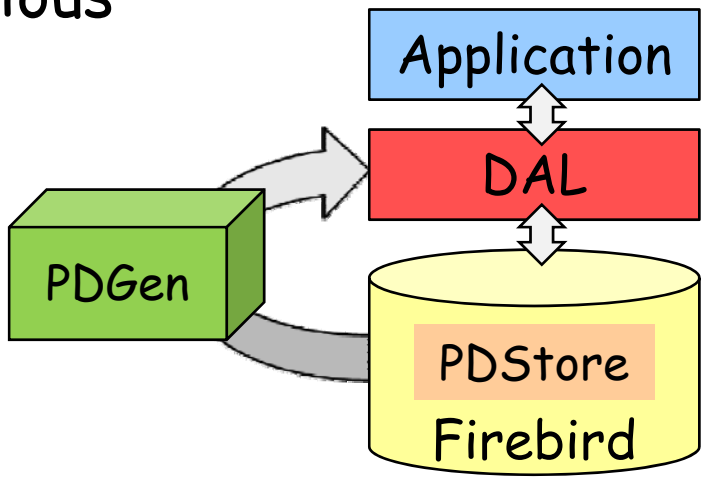
```
<hibernate-mapping>
  <class name="Customer" table="Customer">
    <id name="id" column="ID"> ... </id>
    <property name="name" type="string"
       column="NAME"/>
</class> </hibernate-mapping>
```




```
session.beginTransaction();
Customer c = new Customer();  c.setName("Joe");
session.save(c);  session.getTransaction().commit();
```

PDStore

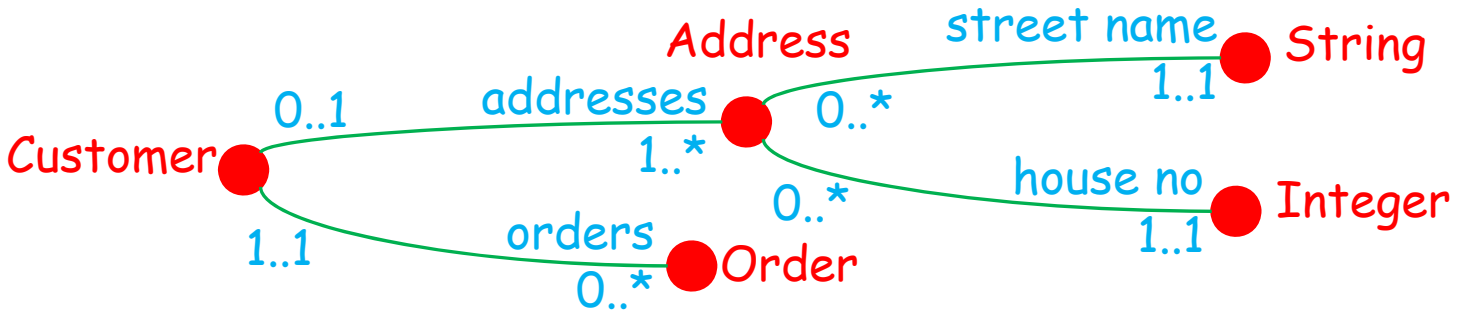
- DB system based the parsimonious data model (PD model)
- Implemented on a on a relational DBMS (Firebird)
- Provides a DAL generator for Java (PDGen)



Advantages:

- All data elements are indentified with GUIDs
  → data from different DBs can easily be merged
- All changes to the data are logged
  → changes can be undone/redone, versioning
- Support for change notification
  → applications can react to changes immediately

Parsimonious Data Model (PD Model)



- Types, Relations and Roles with Multiplicities
- Types are sets of elements
  - Primitive types contain values like strings, ints
  - Complex types contain GUIDs (e.g. for customers)
- Each relation has exactly two roles (one each end)
- Roles may have a name, e.g. "orders", but need not
- Each role has a minimum and a maximum multiplicity
- Types contain instances, relations contain links

Globally Unique Identifiers (GUIDs)

- A identifier that is globally unique (nothing else in the world has the same identifier)
- Consists of 16 bytes
- GUIDs can be generated using the network card (MAC) address of a computer and a timestamp

In PDStore:

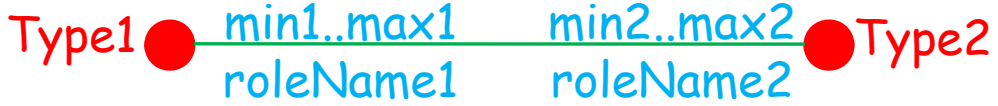
- We can get GUIDs by using the GUIDGen class (just run it and it spits out a list of new GUIDs)
- GUIDs are represented as 32 hex digits, e.g. `66bf14821704dc11b933e6037c01b18f`
- All instances of complex types have GUIDs as IDs

Creating a PDStore Data Model

Create an SQL script in a text file with the following:

1. Connect to the database:
```
CONNECT 'pdstore.fdb' user 'sysdba' password 'masterkey';
```
2. Create a model first:
```
execute procedure create_model('model guid', 'model name');
```

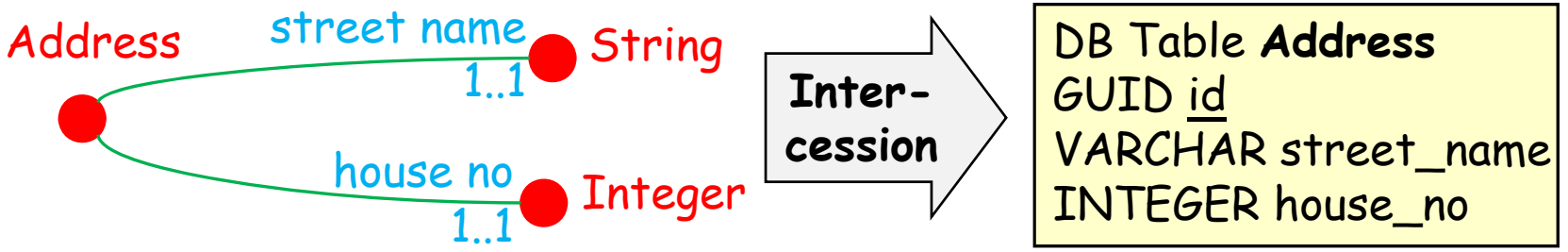
Now go through the elements of the model:



3. For each type: (primitve types are defined in pdstore.sql)
```
execute procedure create_type('type guid',
'model guid', 'type name', null);
```
4. For each relation:
```
execute procedure create_relation(
'role1 guid','type1 guid',min1,max1,'roleName1',
'role2 guid','type2 guid',min2,max2,'roleName2');
```
5. Add a commit;

Creating a PDStore Data Model Cont.

6. After creating a model with types and relations, tell PDStore to create all the corresponding DB tables:
   `execute procedure intercession('model guid');`



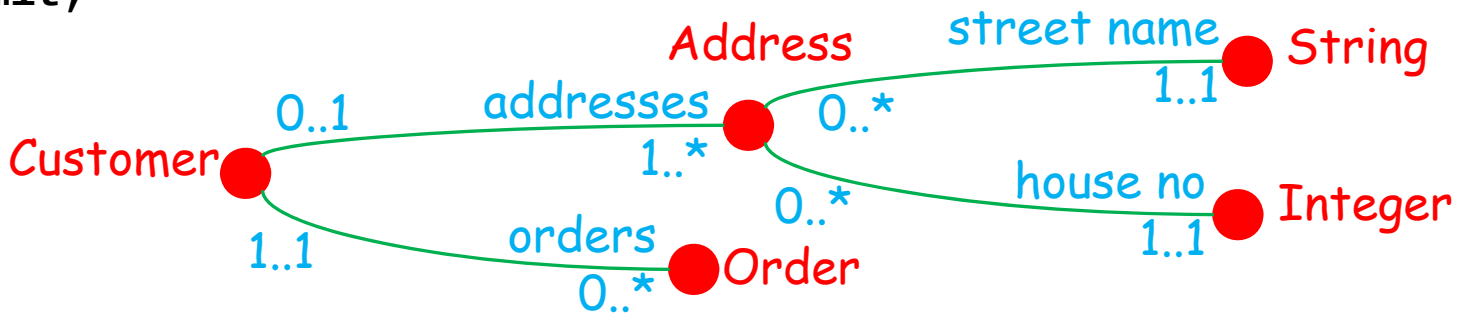
7. Add another `commit;`
8. Add the SQL script (*e.g.* `mymodel.sql`) to `reset-pdstore.bat`

```
del .\pdstore.fdb
fsql\fsql -i pdstore.sql 2> pdstore-errors.txt
fsql\fsql -i mymodel.sql 2> mymodel-errors.txt
```

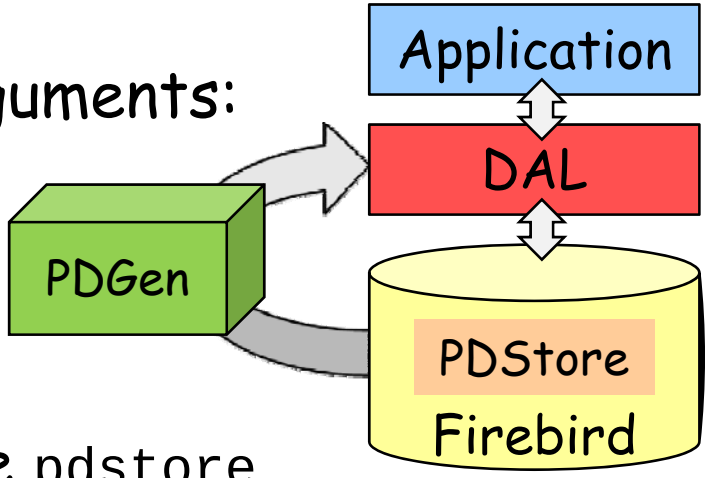
9. Start Development → Dev. Env. → Firebird → Fb. Guardian; Run `reset-pdstore.bat` and you get your model in a fresh database in file `pdstore.fdb`

Example Model: mymodel.sql

```
CONNECT 'pdstore.fdb' user 'sysdba' password 'masterkey';
execute procedure
   create_model('4ef3e2dab0b9dd11b1bff11d9e19f111', 'My Model');
execute procedure create_type('09ca301f191edd11ad8da2fa74ba0698',
   '4ef3e2dab0b9dd11b1bff11d9e19f111', 'Customer', null);
execute procedure create_type('10ca301f191edd11ad8da2fa74ba0698',
   '4ef3e2dab0b9dd11b1bff11d9e19f111', 'Address', null);
execute procedure create_relation(
   '57f3e2dab0b9dd11…', '09ca301f191edd1…', 0, 1, null,
   '58f3e2dab0b9dd11…', '108a986c4062db1…', 1, null, 'addresses');
/* do the same for all other types and relations */
commit;
execute procedure intercession('4ef3e2dab0b9dd11b1bff11d9…');
commit;
```



Generating a DAL with PDGen



- Run PDGen with the following arguments:
  1. Model name (`"My Model"`)
  2. Source root (`src`)

  e.g. `java PDGen "My Model" src`
- All DAL classes will be in package `pdstore`
- For all types `X` in the model it will generate class `PDX`
- DAL classes will have getters and setters for all the named accessible roles, e.g. class `PDAddress` will have

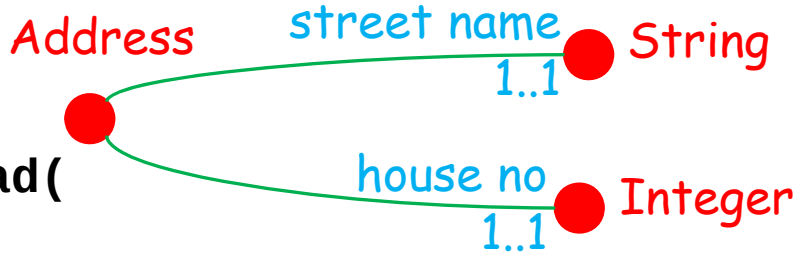
```
String getStreetName()
void   setStreetName(String streetName)
```

and class **`PDCustomer`** will have

```
Set<PDAddress>  getAddresses()
void   addAddresses(PDAddress addresses)
void   removeAddresses(PDAddress addresses)
```

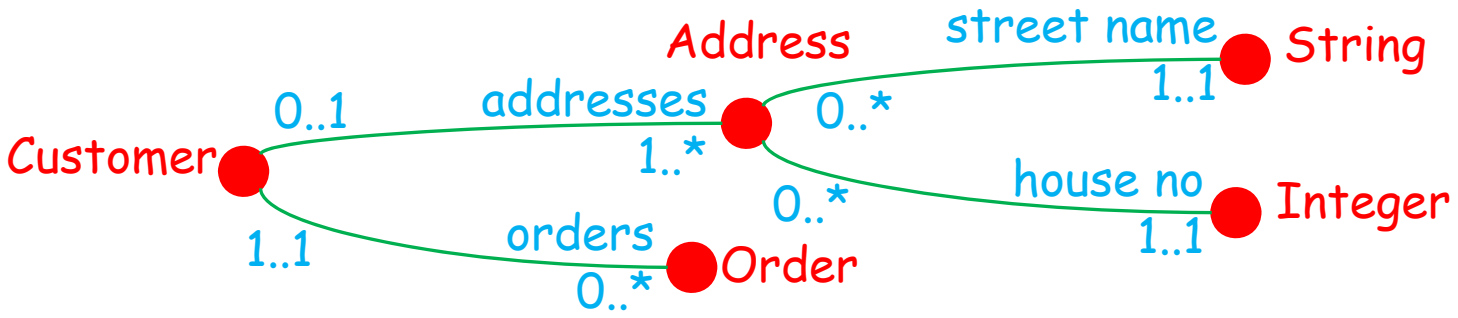
Using the Generated DAL

```
import pdstore.*;  // at the beginning of your file

// create a new cache that is connected to the DB
PDCache cache = new PDCache(
   "jdbc:firebirdsql:local:.\\pdstore.fdb",
   "sysdba", "masterkey");

// load an instance into memory
PDAddress a = (PDAddress) cache.load(
   PDAddress.typeId, "My Address");

a.setStreetName("Symond Street");       // use setter
System.out.println(a.getStreetName());  // use getter
a.setHouseNo(108);

cache.commit();  // make changes permanent
```



Using the Generated DAL Cont.

```
// create a new instance
PDCustomer c = (PDCustomer) cache.newInstance(
   PDCustomer.typeId);
// every instance can have a name (initially it is NULL)
System.out.println(c.getName());
c.setName("Joe"); // name can be used for loading instance
// use getters and setters for roles with multiple links
Set<PDAddress> addresses = c.getAddresses();  // empty
c.addAddresses(a);     // now there is one address
c.removeAddresses(a);  // and now it is gone
cache.commit();  // make changes permanent
```

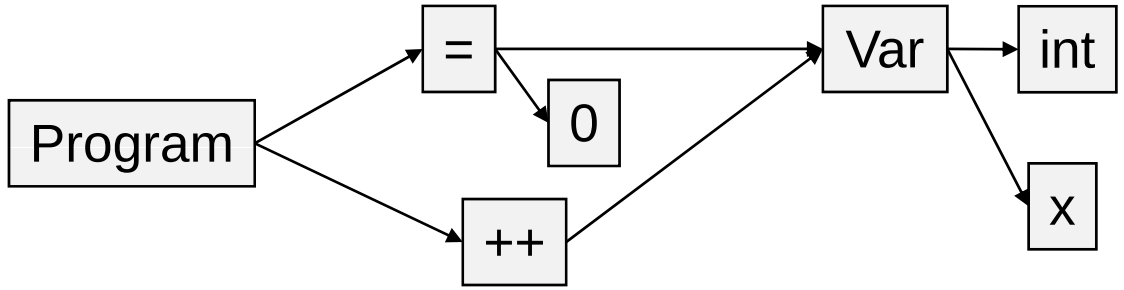


Assignment 2 Project

Structured Source Code Representation

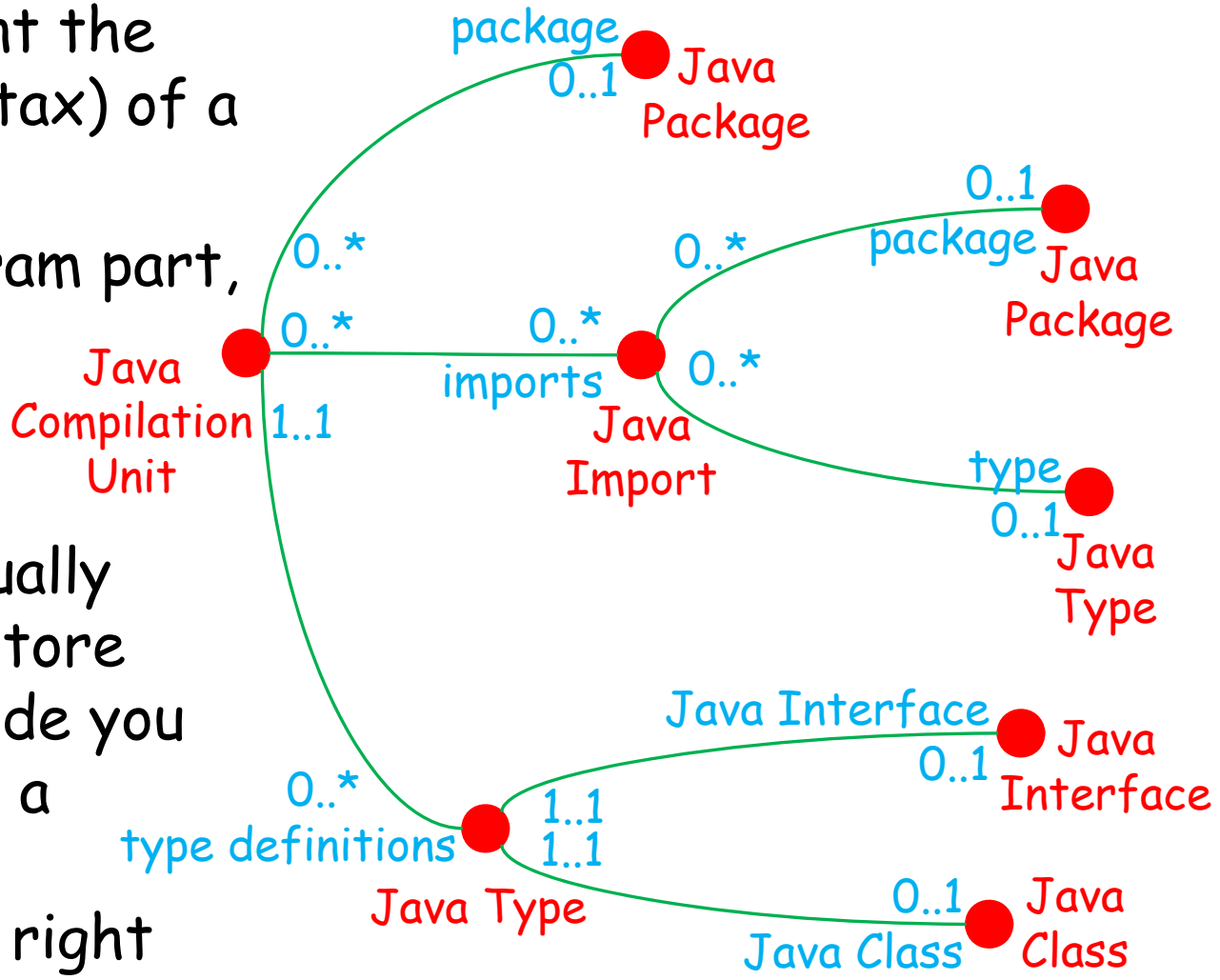
- Most tools deal with source code in a textual representation
  - Unnecessary lexical details; error prone because depends on every single character ("untyped")
  - Linear in contrast to inherent tree-like structure, thus hard to process (parser required) and inflexible
- Idea: use structured, typed AST representation for source code
- Many modern IDEs use such an approach internally
  - Source code is parsed (often while typing) and stored as AST
  - AST is used for navigation, editing, transformation, analysis
- Structured representation enables new functionality
  - Enhanced retrieval: search, aggregation, elision
  - Typed editing (inherent prevention of syntactic errors)

```
int x = 0;
x++;
```



Abstract Syntax Tree (AST)

- ASTs represent the structure (syntax) of a program
- For each program part, the AST contains a node
- The AST is usually typed, e.g. to store Java source code you need to create a PD model like the one on the right side



Summary

Today's Summary

- Data Access Layers (DALs) enables the use of OO classes to read and write from/to the DB
  - Can be written manually
  - Can be generated with a DAL generator
  - Can be generic, i.e. able to handle any data given a mapping
- PDStore is a DB system that is based on GUIDs
  - Types, Relations and Roles with Multiplicities
  - Models are specified in an SQL script
  - Has a DAL generator for Java

Quiz

1. What is a DAL?
2. Describe the advantages and disadvantages of the different ways to create DALs.
3. What is a GUID and why is it useful?
4. How are data models specified in PDStore?