

ADO.NET

Objective

- **Introduce ADO.NET and SQL Server interaction**
 - connection
 - command
 - data reader
 - stored procedure
 - disconnected data set
 - database independent coding



Overview

- **ADO.NET provides managed types for database access**
 - generic types in `System.Data` namespace
 - SQL Server types in `System.Data.SqlClient` namespace
 - other data providers also supported



Connection parameters

- **Must specify several pieces of information to connect**
 - server
 - database
 - authentication credentials
- **Exact connection details differ for different providers**



Server

- **Use server parameter to specify server for SQL Server**
 - passed in connect string
 - use "." or "localhost" to connect to local database

specify
server →

```
string connectionString = "Server=localhost;...";
```



Database

- Use Database parameter to specify database for SQL Server
 - passed in connect string

specify
database →

```
string connectionString = "Database=pubs;...";
```



Authentication

- **Two ways to authenticate a client connection for SQL Server**
 - Windows authentication uses Windows user information
 - **Integrated Security** set to **SSPI** in connect string
 - SQL Server authentication uses SQL Server user information
 - **User ID** and **Password** passed in connect string

use Windows
information →

```
string connectionString = "Integrated Security=SSPI;...";
```

use SQL
information →

```
string connectionString = "User ID=Joe; Password=lobster;...";
```



Connecting

- Use `SqlConnection` to connect to SQL Server
 - create object
 - can pass to constructor
 - can set after creation using `ConnectionString` property
 - call `Open` method



parameters →

```
string cs = "server=.;Integrated Security=SSPI;database=pubs";
```

create →

```
SqlConnection connection = new SqlConnection(cs);
```

open →

```
connection.Open();
```

```
...
```



Disconnecting

- **Close SqlConnection when finished**
 - can call either `Close` or `Dispose` method
 - typical to place call in `finally` block or `using` statement

```
static void Main()
{
    SqlConnection connection = null;

    try
    {
        ...
        connection.Open();
        ...
    }
    finally
    {
        connection.Dispose();
    }
}
```

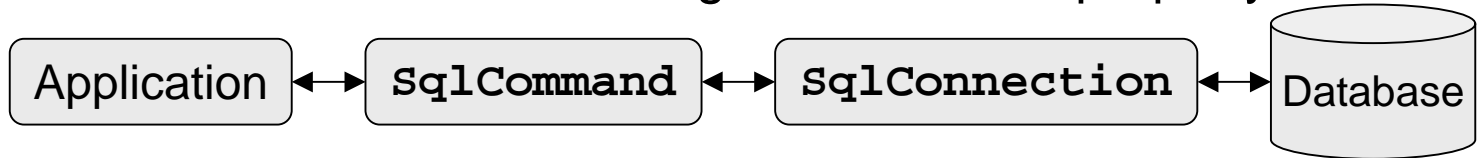
open →

close →



Command setup

- Use `SqlCommand` to execute command
 - must specify command text
 - can pass to constructor
 - can set after creation using `CommandText` property
 - must specify connection to use
 - can pass to constructor
 - can set after creation using `Connection` property



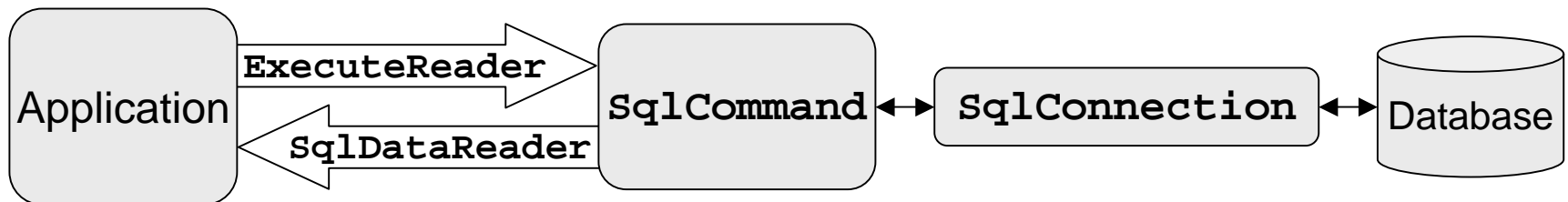
```
SqlConnection connection = new SqlConnection(...);  
...  
string text = "select * from authors";  
  
SqlCommand command = new SqlCommand(text, connection);  
...
```

create command →



SqlCommand.ExecuteReader

- Use `ExecuteReader` when result set expected
 - returned data placed in `SqlDataReader` object
 - reader provides forward-only access to data
 - multiple results supported using `NextResult`
 - throws `Exception` if command fails



```
string      text      = "select * from authors";
SqlConnection connection = new SqlConnection(...);
SqlCommand  command   = new SqlCommand(text, connection);
...
SqlDataReader reader = command.ExecuteReader();
```

↑
capture returned data

↑
execute command



SqlDataReader data access

- **Two main ways to access rows of result set**
 - use `foreach` to traverse rows of `IDataRecord` objects
 - use `while` loop with `Read` to manually advance through rows
- **Three main ways to access columns of a row**
 - index by column ordinal or name
 - pass column index to `getXXX` methods
 - use `for` loop with `FieldCount` to access each column in turn

loop through rows →

access data in row
using indexers →

access data in row
using get method →

```
static void Display(SqlDataReader reader)
{
    while (reader.Read())
    {
        string last = (string)reader["au_lname"];
        string first = (string)reader[2];

        string zip = reader.GetString(7);
        ...
    }
}
```



SqlDataReader Close

- **Call Close when finished with SqlDataReader**
 - releases connection (which can then be reused)
 - can not access contained data after closing

```
string      text      = "select * from authors";
SqlConnection connection = new SqlConnection(...);
SqlCommand  command   = new SqlCommand(text, connection);
...
SqlDataReader reader = command.ExecuteReader();
...
reader.Close();
```

↑
close reader
when finished



SqlCommand ExecuteNonQuery

- Use `ExecuteNonQuery` when no data will be returned
 - returns an `int` specifying number of rows affected



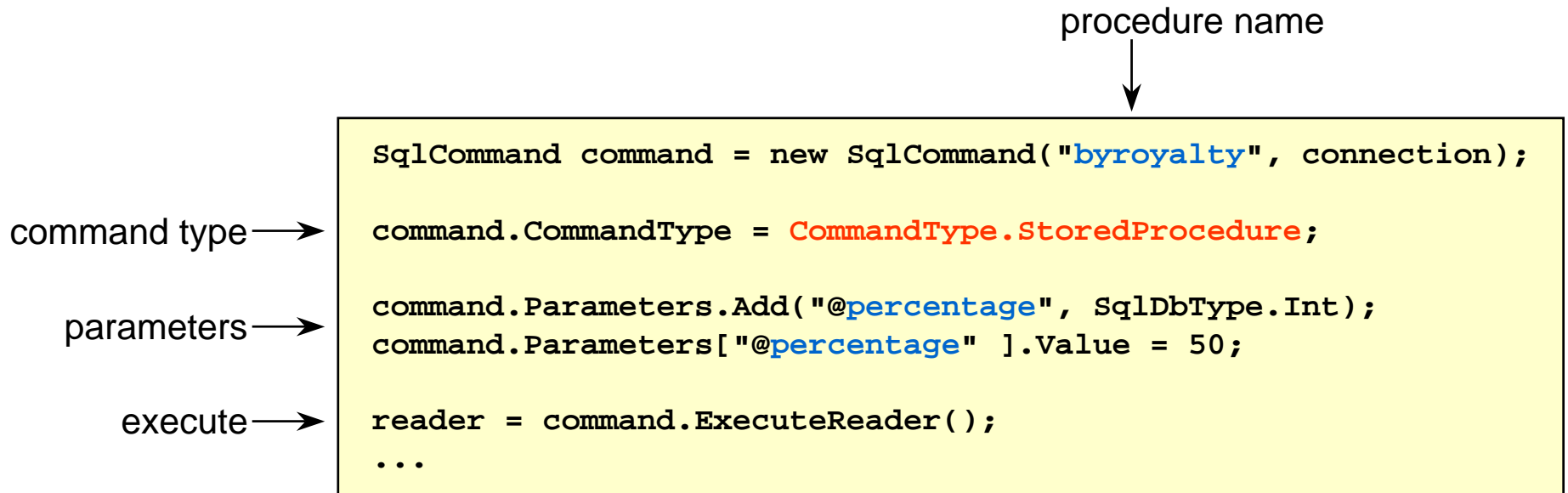
```
string text = "insert into authors "  
    + "(au_id, au_lname, au_fname, contract) values "  
    + "('111-11-1111', 'Adams', 'Mark', 1)";  
  
SqlCommand command = new SqlCommand(text, connection);  
  
int rowsAffected = command.ExecuteNonQuery();  
...
```

execute
command →



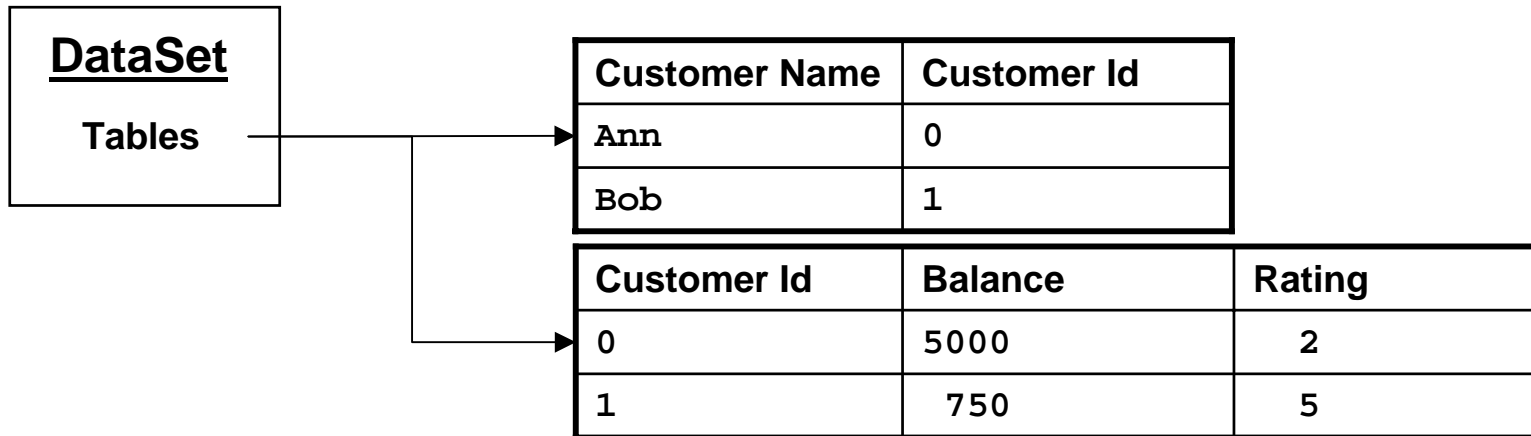
Stored procedure

- Use `SqlCommand` to execute stored procedure
 - set `CommandType` property to `StoredProcedure`
 - set `CommandText` property to procedure name
 - pass parameters in `Parameters` collection
 - call `ExecuteReader`



Disconnected data in ADO.NET

- **Can model in-memory cache of data**
 - tables, relations, rows, columns etc.
 - disconnected and independent of data source



DataSet

- **DataSet class models disconnected data set**
 - has collection property of **DataTable** objects

tables in data set →

```
public class DataSet ...  
{  
    public DataTableCollection Tables { get; }  
    ...  
}
```



DataTable

- **DataTable is in-memory model of a table**
 - has rows, columns, etc.

rows and columns
currently in table →

```
public class DataTable ...
{
    public DataRowCollection Rows { get; }
    public DataColumnCollection Columns { get; }
    ...
}
```



DataRow

- **DataRow is in-memory model of row inside DataTable**
 - several ways to access column data
 - rows are generated by tables, not created directly

access column data
by name or number →

all columns in row →

```
public class DataRow ...
{
    public object this[string] { get; set; }
    public object this[int] { get; set; }

    public object[] ItemArray { get; set; }
    ...
}
```



DataColumn

- DataColumn models column of DataTable
 - specify name and data type when creating

```
public class DataColumn ...  
{  
    public DataColumn(string name, Type type);  
    ...  
}
```

↑
name of
column

↑
Type object for
type of column data



Generating DataSet

- **Two main ways to create DataSet**
 - fill from existing data source
 - manually define structure and fill with data



Fill DataSet from source

- Can fill DataSet from existing source such as database
 - use SqlDataAdapter and its Fill method

create adapter →
create DataSet →
use adapter to
fill DataSet →

```
string      text = "select * from authors";  
SqlConnection conn;  
...  
SqlDataAdapter adapter = new SqlDataAdapter(text, conn);  
  
DataSet authors = new DataSet();  
  
adapter.Fill(authors);
```



Traverse DataSet

- DataSet three levels of data inside
 - set of contained tables
 - rows in each table
 - column values in each row

traverse DataSet →

```
foreach (DataTable table in myDataSet.Tables)
{
    foreach (DataRow row in table.Rows)
    {
        foreach (object data in row.ItemArray)
        {
            // process column value
        }
    }
}
```



Manual creation of DataTable

- **Can manually create DataTable**
 - define table structure
 - create rows, add rows to tables, fill with data

create table → `DataTable customers = new DataTable("Customers");`

define columns → `customers.Columns.Add("Name", typeof(string));`
`customers.Columns.Add("Id", typeof(Int32));`

create row → `DataRow row = customers.NewRow();`

populate row using indexer → `row[0] = "Ann";`

add row to table → `row[1] = 0;`
`customers.Rows.Add(row);`

...



Manual creation of DataSet

- Can manually create DataSet
 - create DataSet object
 - create tables and add to set

```
create table → DataTable customers = new DataTable("Customers");  
              ...  
create DataSet → DataSet data = new DataSet();  
add table to set → data.Tables.Add(customers);  
                  ...
```



Updating database through DataAdapter

- Can update database after changing DataSet
 - use SqlCommandBuilder to create needed SQL commands
 - use Update method of DataAdapter to send changes

specify source table
when filling data set



```
string text = "select * from authors";  
  
SqlDataAdapter adapter = new SqlDataAdapter(text, conn);  
  
DataSet ds = new DataSet();  
  
adapter.Fill(ds, "authors");
```

modify DataSet



```
ds.Tables[0].Rows[4][2] = "Bob";
```

attach builder



```
SqlCommandBuilder b = new SqlCommandBuilder(adapter);
```

update database

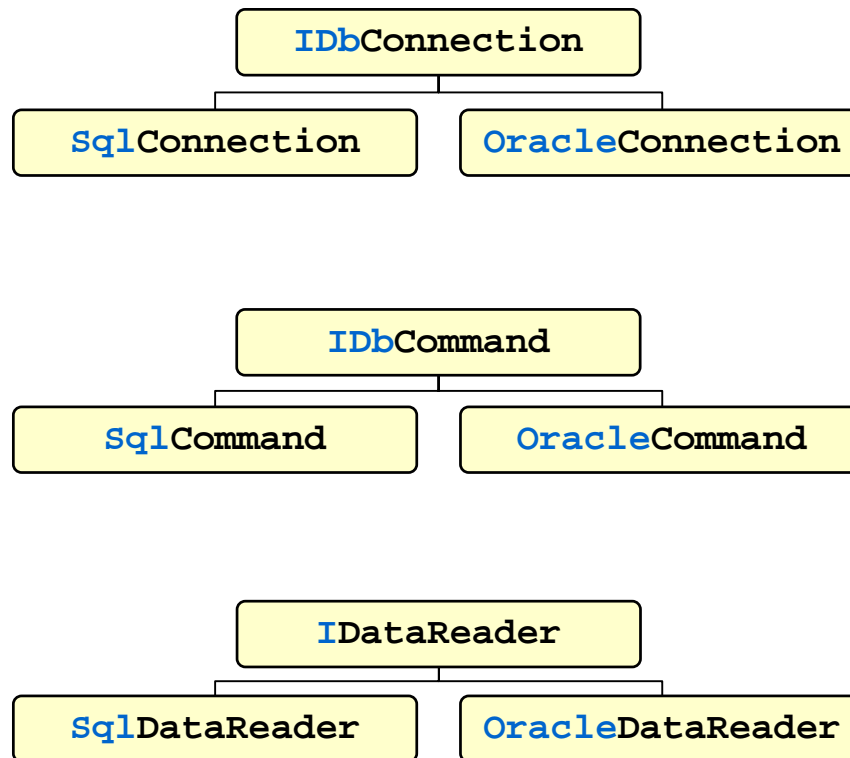


```
adapter.Update(ds, "authors");
```



Database independent coding

- **Separate classes provided to access various databases**
 - implement same interfaces



Using interface reference

- Use interface reference to write more generic code

```
string text = "select * from authors";  
  
IDbConnection connection = new SqlConnection(...);  
...  
IDataReader reader = command.ExecuteReader();
```

↑
database independent
interface references

↑
database specific
classes



Summary

- **ADO.NET provides managed objects to do database access**
 - can execute queries, run stored procedures, etc.
 - can operate in connected or disconnected mode
 - generic interfaces help hide database specific types

