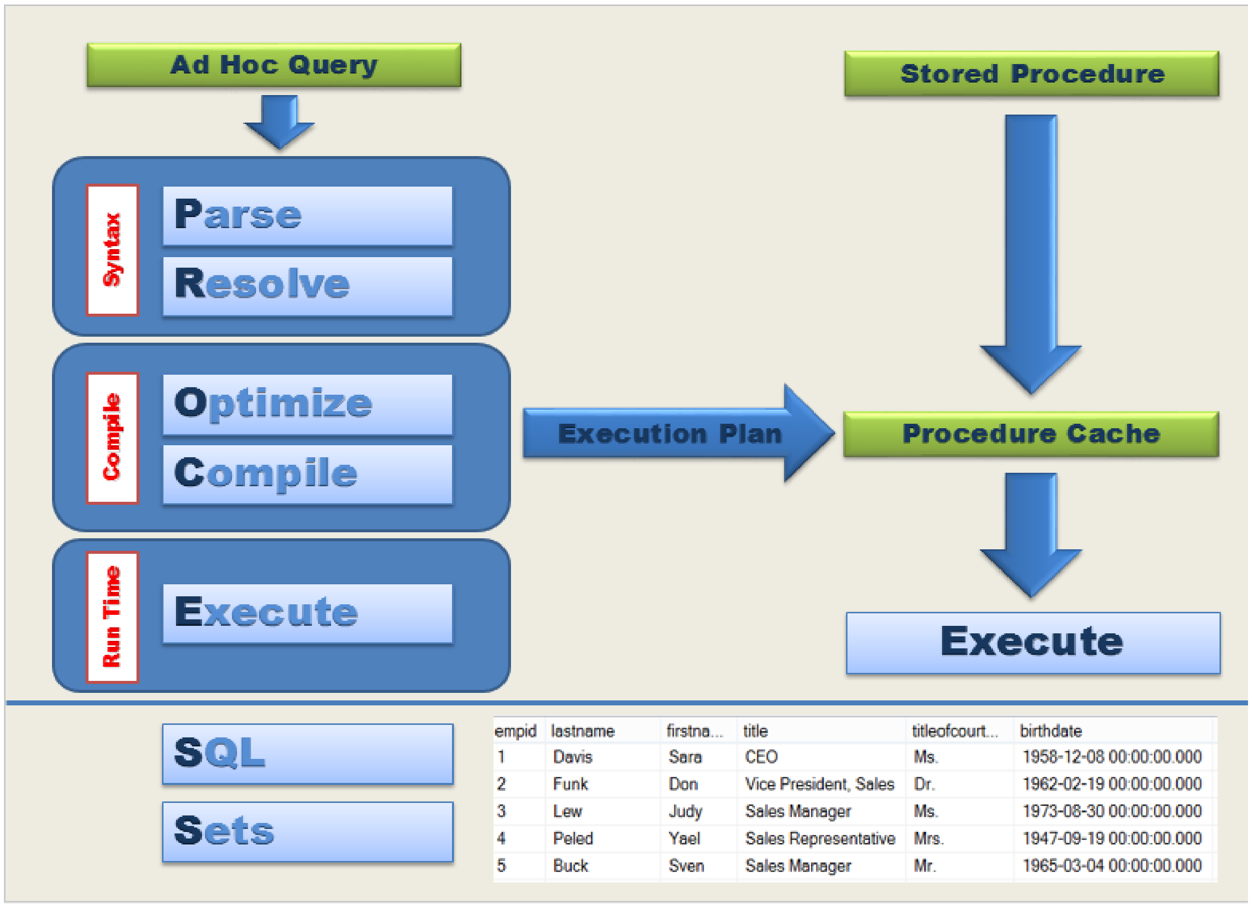
SQL Query Processing



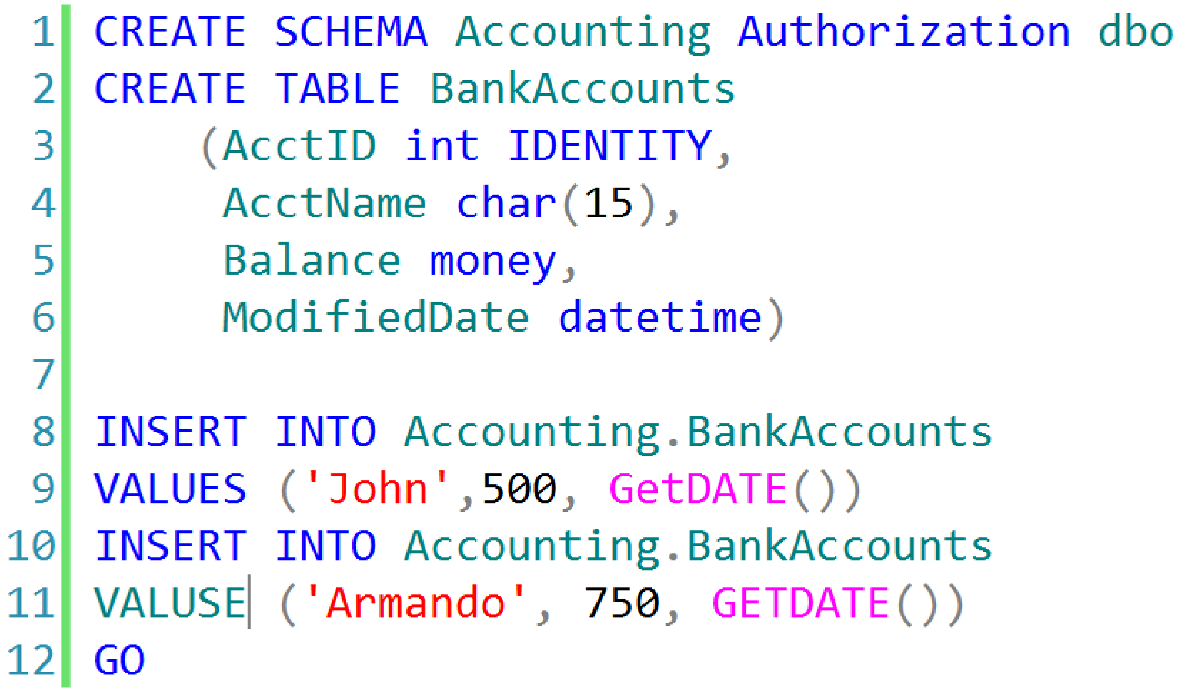
empid	lastname	firstna...	title	titleofcourt...	birthdate
1	Davis	Sara	CEO	Ms.	1958-12-08 00:00:00.000
2	Funk	Don	Vice President, Sales	Dr.	1962-02-19 00:00:00.000
3	Lew	Judy	Sales Manager	Ms.	1973-08-30 00:00:00.000
4	Peled	Yael	Sales Representative	Mrs.	1947-09-19 00:00:00.000
5	Buck	Sven	Sales Manager	Mr.	1965-03-04 00:00:00.000

The first time through running an Ad Hoc query or Stored Procedure, SQL Server will go through each of the following steps.

1. The first step is to **Parse** the statement into keywords, expressions, and operators. This is where the syntax of your query statement is checked for accuracy.
2. The second step is to **Resolve** objects (Tables, Views, Columns, etc.) to see if they exist. Both the first and second step is where you will find syntax errors. In most cases, these errors are caused by misspellings or putting commas in the wrong place.
3. The third step is to **Optimize** the query. This is where the query optimizer will find different ways of locating data from your tables based on available indexes and/or statistics. Once it discovers the fastest steps to retrieve your result set using the least amount of resources, it will create an Execution Plan. Errors would only occur during this stage if there is a lack of hardware resources. Most compile errors happen at the next step.
4. The fourth step is to **Compile** the Execution Plan to store in the Procedure Cache for future use.
5. Finally, the Execution Plan will **Execute** and hopefully return the desired **SQL Sets.** This is where Run-Time errors will occur that need to be handled using Exception Handling.

Additional submissions of the query, such as from a Stored Procedure, will check the Procedure Cache for existing or similar Execution Plans that could be used for the query. If this is the case, the existing Execution Plan will be used to retrieve your SQL Sets.

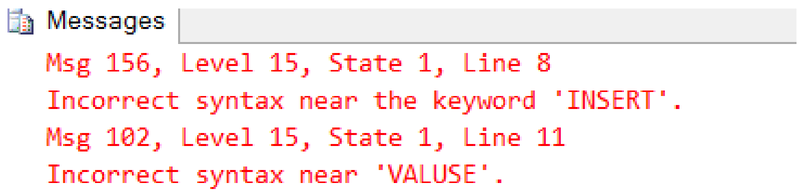
Working with Batches



```
 1  CREATE SCHEMA Accounting Authorization dbo
 2  CREATE TABLE BankAccounts
 3      (AcctID int IDENTITY,
 4       AcctName char(15),
 5       Balance money,
 6       ModifiedDate datetime)
 7
 8  INSERT INTO Accounting.BankAccounts
 9  VALUES ('John',500, GetDATE())
10  INSERT INTO Accounting.BankAccounts
11  VALUSE ('Armando', 750, GETDATE())
12  GO
```

To begin our example we will need to run both the Create Schema and Create Table to hold sample data. Then we will use two Insert statements.

Notice we do not have a GO statement on line 7 to divide the CREATE statements from the INSERT statements into separate batches. This will produce the following errors.

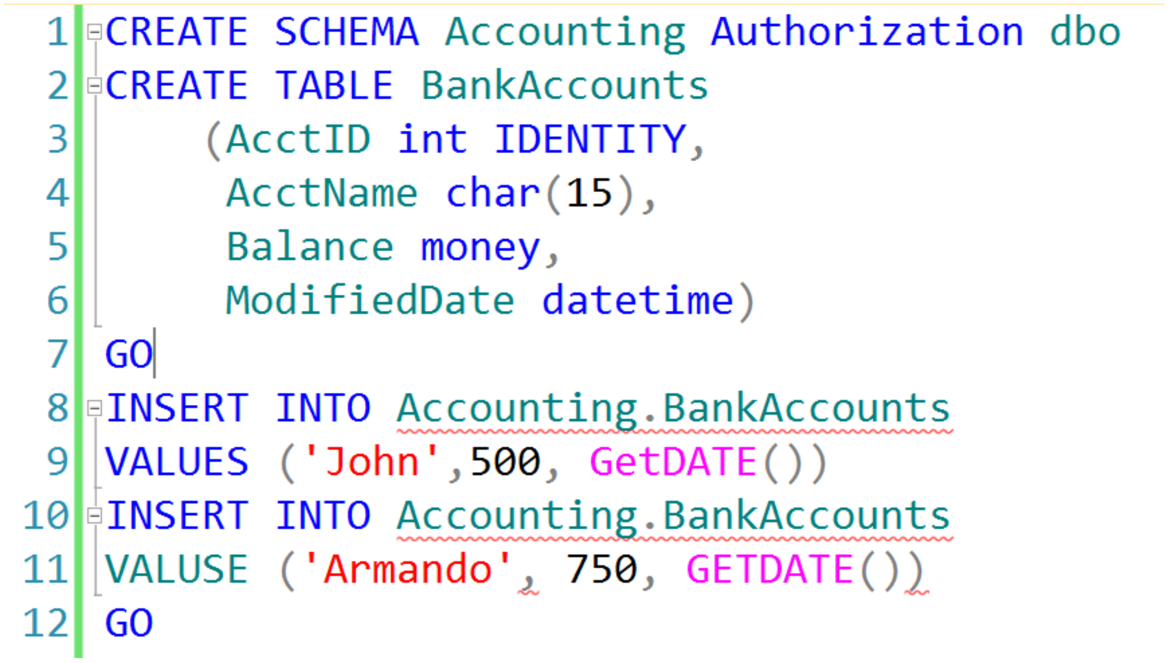


```
Messages
  Msg 156, Level 15, State 1, Line 8
  Incorrect syntax near the keyword 'INSERT'.
  Msg 102, Level 15, State 1, Line 11
  Incorrect syntax near 'VALUSE'.
```

These errors happen because batches are used to divide code into sections for Parsing Syntax and Resolving Object names. The Msg 156 error is an example of an object resolution error and occurs because SQL has not yet created the SCHEMA or TABLE objects that the INSERT statements needed to add data. In addition, DDL statements such as the CREATE SCHEMA statement need to be in their own batches and will generate a syntax error as well. The Msg 102 error is also an example of a parsing syntax error and occurred because the keyword VALUES was misspelled.

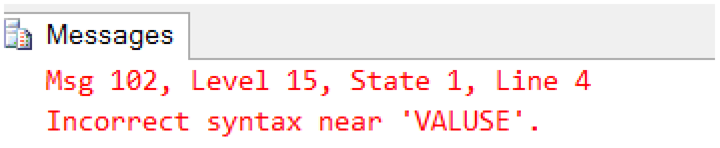
Separating Batches

We will add the GO statement to line 7 to divide the two code sections into separate batches. Notice we will not correct the misspelled keyword VALUES.



```
 1 CREATE SCHEMA Accounting Authorization dbo
 2 CREATE TABLE BankAccounts
 3      (AcctID int IDENTITY,
 4       AcctName char(15),
 5       Balance money,
 6       ModifiedDate datetime)
 7 GO
 8 INSERT INTO Accounting.BankAccounts
 9 VALUES ('John',500, GetDATE())
10 INSERT INTO Accounting.BankAccounts
11 VALUSE ('Armando', 750, GETDATE())
12 GO
```

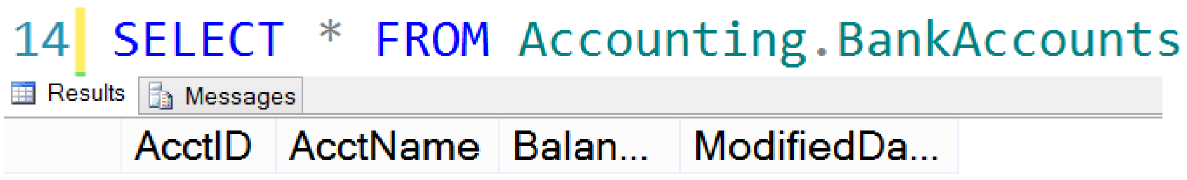
When we run this code we will still get the Parsing Syntax error.



Messages

```
Msg 102, Level 15, State 1, Line 4
Incorrect syntax near 'VALUSE'.
```

However, our Accounting Schema and BankAccounts Table have both been created. This is because the CREATE statements were in their own batch and was able to run without error. Also, notice that when we run a SELECT statement on the table no records will be displayed.



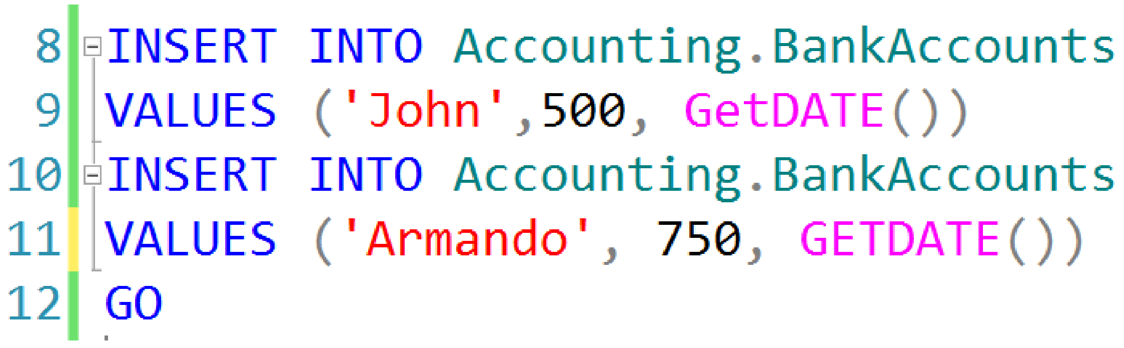
```
14 SELECT * FROM Accounting.BankAccounts
```

Results | Messages

	AcctID	AcctName	Balan...	ModifiedDa...

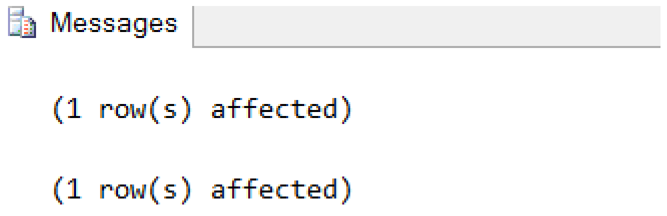
Syntax Error in Batches

Because there was a syntax error in the batch that included both INSERT statements the entire batch failed. Next we will fix the misspelling of the keyword VALUES on line 11.



```
 8 INSERT INTO Accounting.BankAccounts
 9 VALUES ('John',500, GetDATE())
10 INSERT INTO Accounting.BankAccounts
11 VALUES ('Armando', 750, GETDATE())
12 GO
```

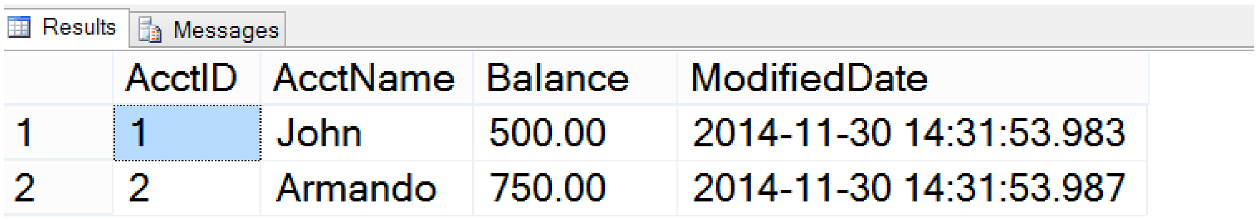
When we run this batch we get the following message. This is because even though the two INSERT statements are in the same batch, they are still two separate implicit transactions.



Messages

```
(1 row(s) affected)

(1 row(s) affected)
```

If we run the SELECT statement again we will see that both INSERT statements ran and added the appropriate records.

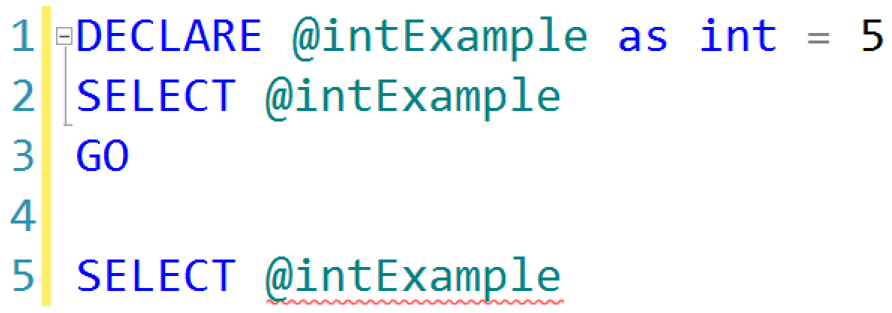


Results | Messages

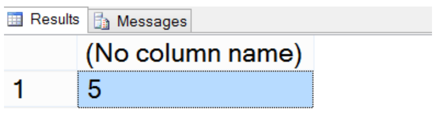
	AcctID	AcctName	Balance	ModifiedDate
1	1	John	500.00	2014-11-30 14:31:53.983
2	2	Armando	750.00	2014-11-30 14:31:53.987

Batches and Variables

One more item of concern with batches is that they not only create a boundary for syntax and parsing, but also they are a boundary for variables. This example will declare and initialize a variable and then SELECT the variable in our first batch.



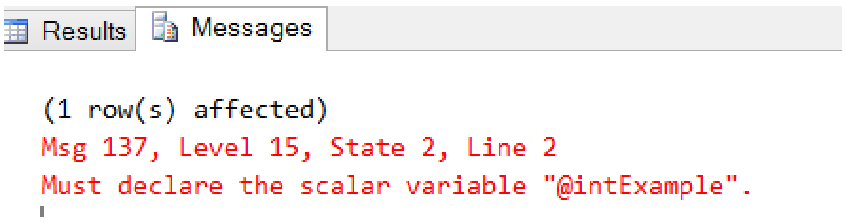
```sql
DECLARE @intExample as int = 5
SELECT @intExample
GO

SELECT @intExample
```



Results | Messages

	(No column name)
1	5

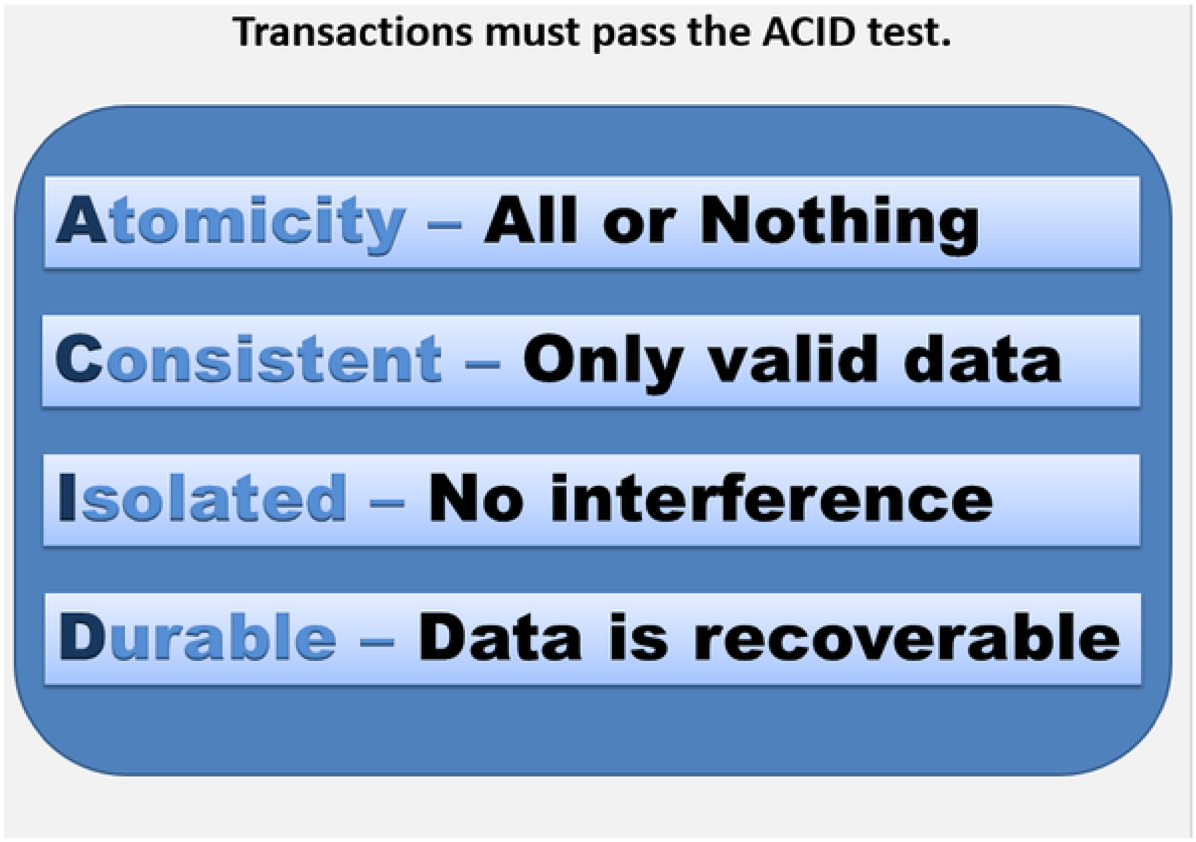
While the first batch ran successfully and displayed the number 5, the second batch will cause the error below to happen. This is because the variable only exists within the batch where it was created.



Results | Messages

```
(1 row(s) affected)
Msg 137, Level 15, State 2, Line 2
Must declare the scalar variable "@intExample".
```

What is a Transaction?



Transactions must pass the ACID test.

Atomicity – All or Nothing

Consistent – Only valid data

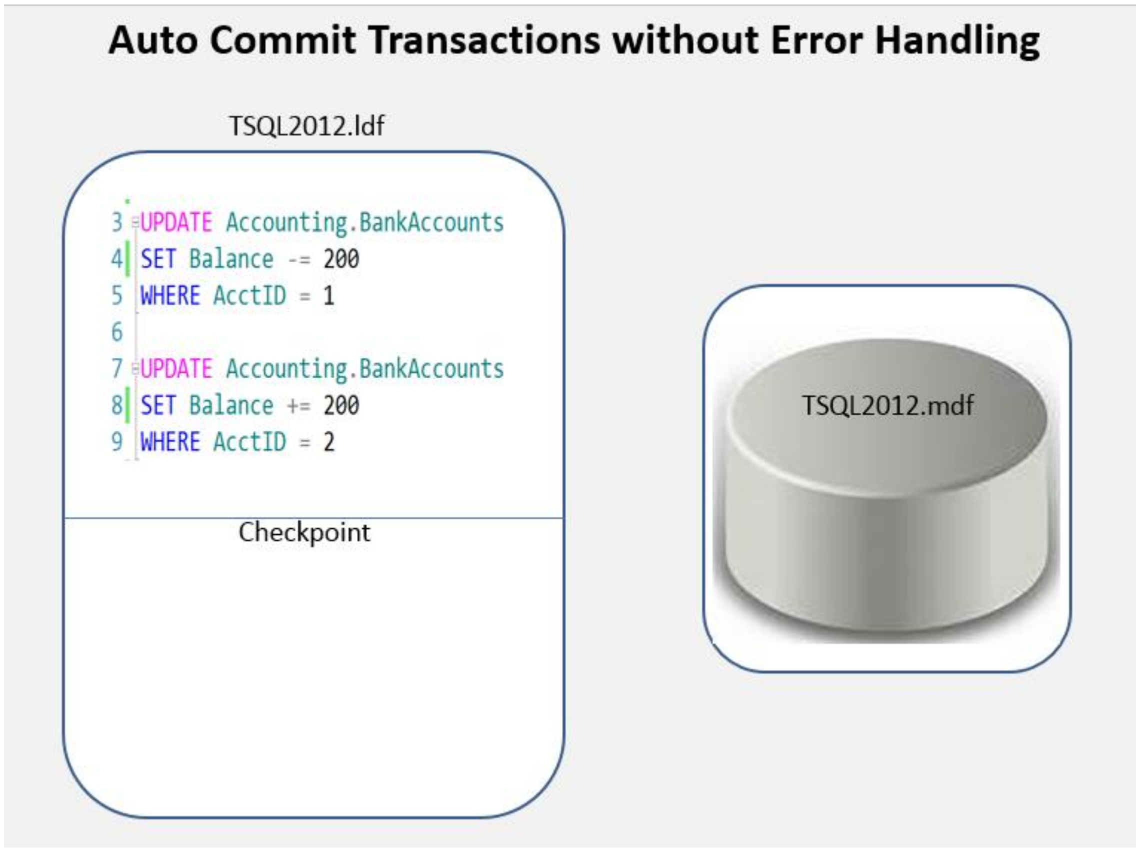
Isolated – No interference

Durable – Data is recoverable

Transactions are used to ensure that a series of statements written to modify data pass the ACID test that enforces the data integrity, accuracy, concurrency, and recoverability of the data in the database. The ACID properties of a transaction are...

- **Atomicity** - This ties several statements together to ensure that a series of statements either all succeed or all fail together.
- **Consistent** - This is actually enforced by using transaction logs to ensure that only valid data that has been committed will be written to the database.
- **Isolated** - This is enforced by using locks to control concurrency. More specifically to make sure that a record being updated by one statement does not interfere with a separate statement trying to modify that same record or records.
- **Durable** - Once again transactions logs are used as a way of recovering data in case of a database failure. This is also controlled by the Recovery model of your database.

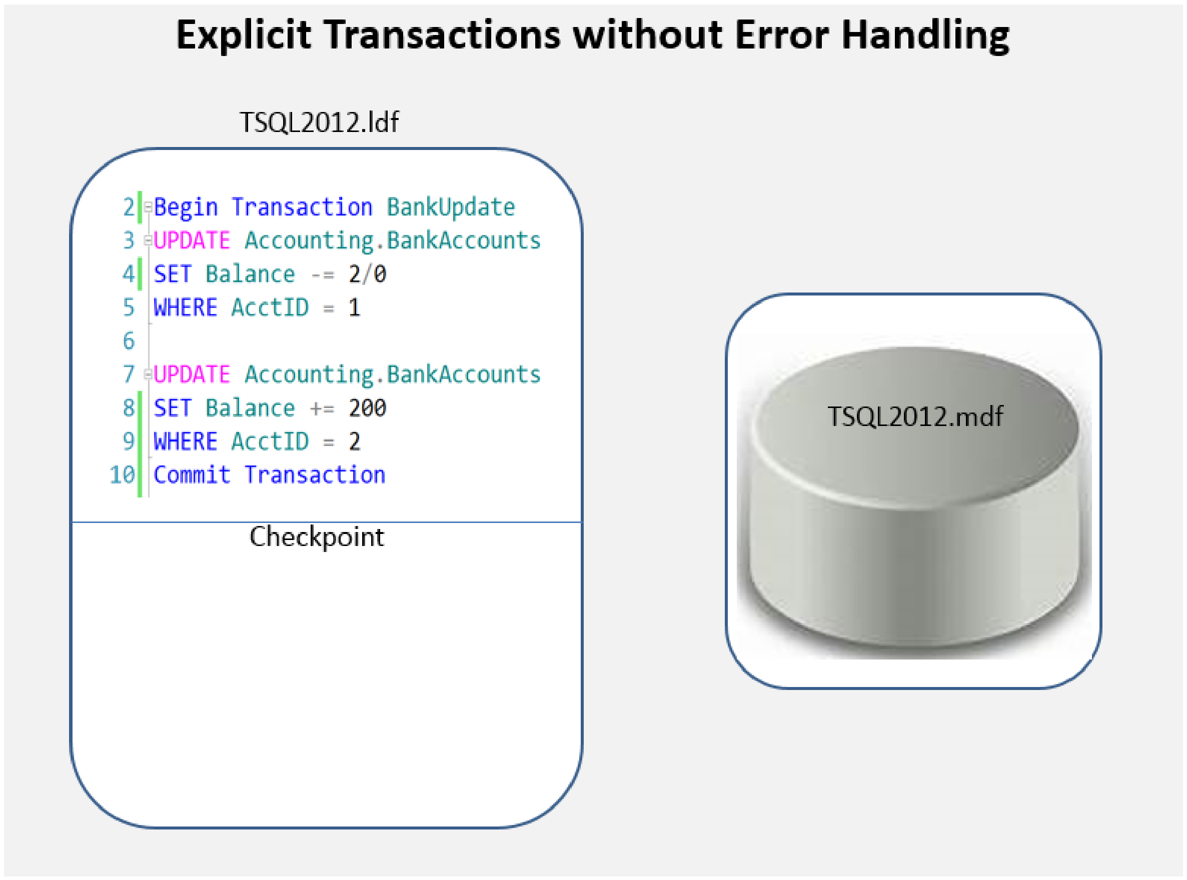
Locks are beyond the scope of this handout. For now, we will look at the theory of using transactions.



```
3 UPDATE Accounting.BankAccounts
4 SET Balance -= 200
5 WHERE AcctID = 1
6
7 UPDATE Accounting.BankAccounts
8 SET Balance += 200
9 WHERE AcctID = 2
```

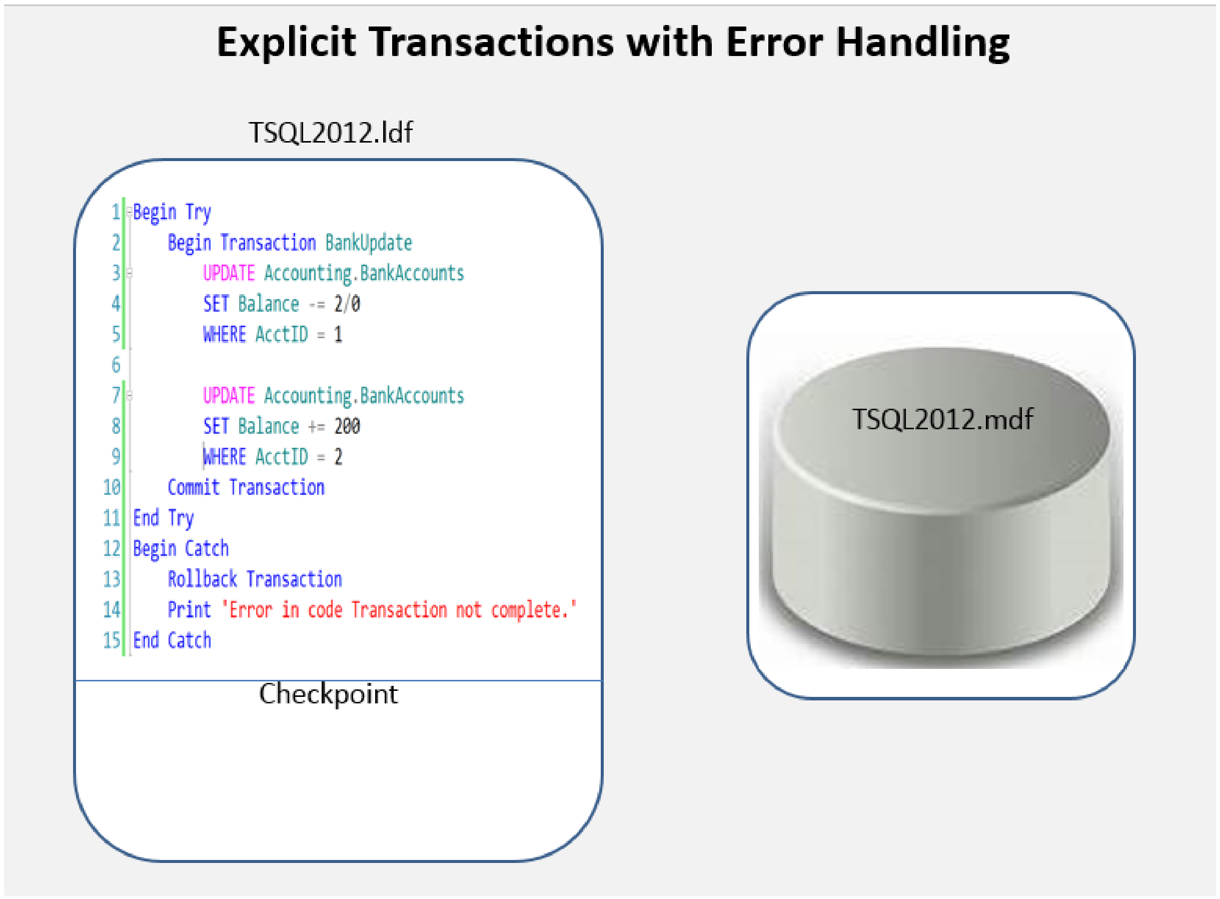
In this example we see two UPDATE statements where $200 is being subtracted from account 1 and then $200 is being added to account 2. Both statements are considered to be two separate Auto Commit Transactions. When the main data file writes a checkpoint to the log file all statements that have been committed at the checkpoint will be written into the database. This is fine if there are not any errors. However, if the UPDATE statement on lines 3, 4, and 5 had an error it would not commit to the database; however the UPDATE statement on lines 7, 8, and 9 would be committed and written to the database. This would cause inconsistent and incorrect data in our database. So we will need to add Explicit Transactions to our code.

Explicit Transactions without Error Handling



```
 2 Begin Transaction BankUpdate
 3 UPDATE Accounting.BankAccounts
 4 SET Balance -= 2/0
 5 WHERE AcctID = 1
 6
 7 UPDATE Accounting.BankAccounts
 8 SET Balance += 200
 9 WHERE AcctID = 2
10 Commit Transaction
```

Here we have used the Begin Transaction and Commit Transaction statements to tie the two statements together. We have also changed the code on line 4 to cause a divide by zero run-time error to happen. When the code runs the first UPDATE statement will cause an error and will not be committed to the database, however, just like before the second UPDATE statement will still be committed. So although we have tied the statements together we still need to add some error handling to manually Rollback the transaction.

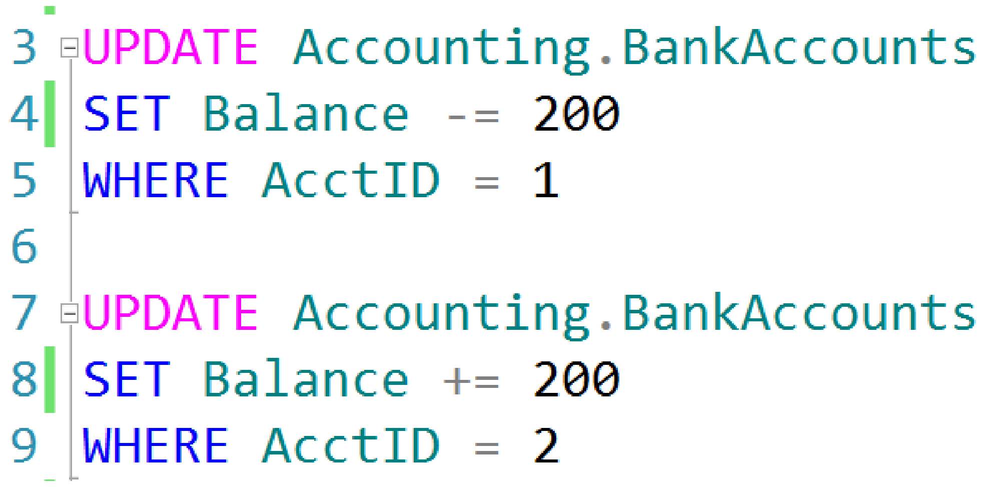


```sql
Begin Try
    Begin Transaction BankUpdate
        UPDATE Accounting.BankAccounts
        SET Balance -= 2/0
        WHERE AcctID = 1

        UPDATE Accounting.BankAccounts
        SET Balance += 200
        WHERE AcctID = 2
    Commit Transaction
End Try
Begin Catch
    Rollback Transaction
    Print 'Error in code Transaction not complete.'
End Catch
```

With the TRY/CATCH error handling added when there is an error anywhere in the code the entire transaction is rollback. So either all the code is successful or none of the code is successful (Atomicity), only data that is valid or committed is written to the database (Consistent), and the data can be recovered from the transaction log (Durable). Again, the Isolated part of a transaction is handled by locks which will be a post in the future.

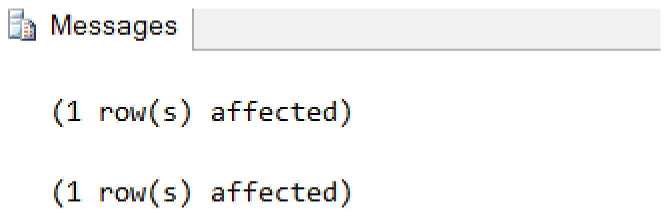
Batches, Transactions and Errors

To take a closer look at enforcing ACID properties on transactions, we will revisit the statements being used when we wanted to transfer $200 from the John account to the Armando account and wanted to make sure that both statements had to be successful or neither statement would commit to the database.



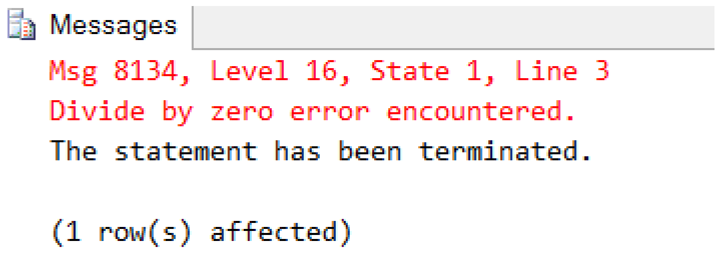
```
3 UPDATE Accounting.BankAccounts
4 SET Balance -= 200
5 WHERE AcctID = 1
6
7 UPDATE Accounting.BankAccounts
8 SET Balance += 200
9 WHERE AcctID = 2
```

This code would run successfully and we would receive the message the each statement was successful.



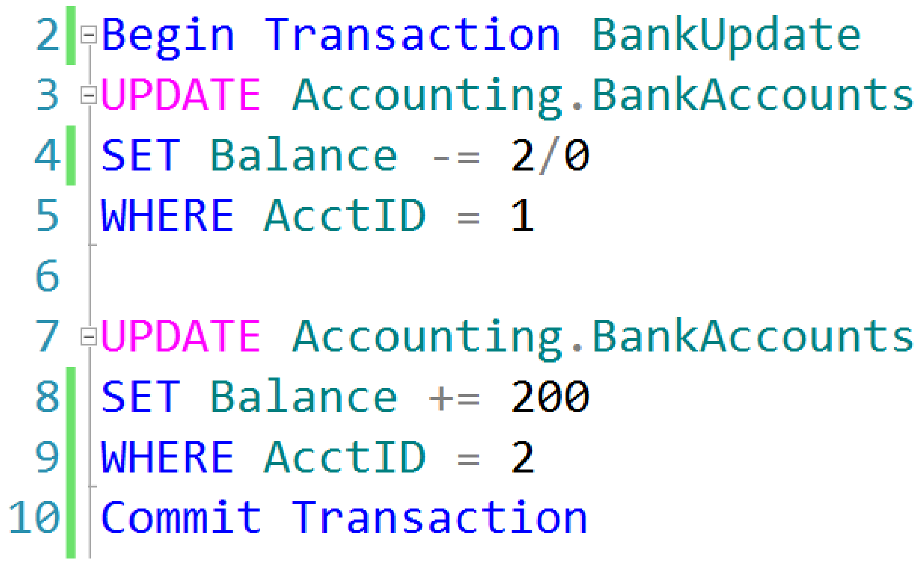
```
Messages

(1 row(s) affected)

(1 row(s) affected)
```

However, what if we made an error in one of the statements? For example, on line 4 we accidentally typed 2/0 instead of 200 which would give a divide by zero run-time error.



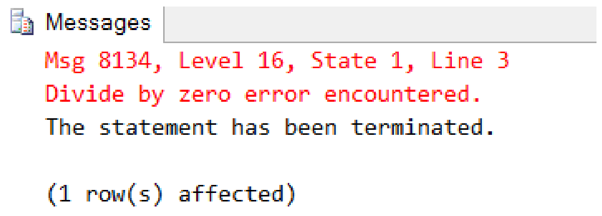
```
Messages
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.
The statement has been terminated.

(1 row(s) affected)
```

Notice that the first update statement failed, but the second update statement still ran successfully. This would lead to inconsistent data in our tables where $200 was added to the Armando account without removing it from the John account. So to ensure that either both statements are successful or neither statements run, we will create an Explicit Transactions by using the Begin Transaction and Commit Transaction statements.



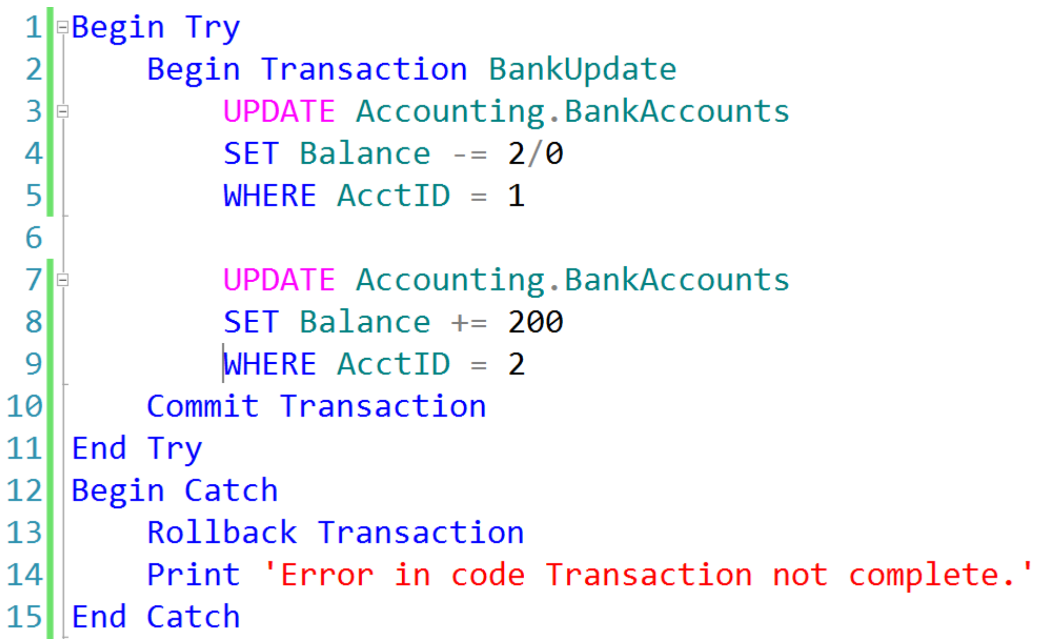
```
 2 Begin Transaction BankUpdate
 3 UPDATE Accounting.BankAccounts
 4 SET Balance -= 2/0
 5 WHERE AcctID = 1
 6
 7 UPDATE Accounting.BankAccounts
 8 SET Balance += 200
 9 WHERE AcctID = 2
10 Commit Transaction
```

However, when we run this code we still get the same message.



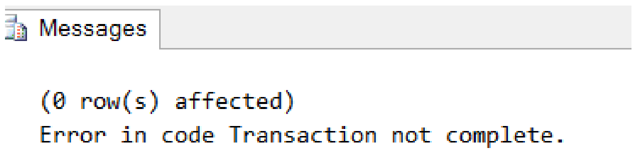
```
Messages
Msg 8134, Level 16, State 1, Line 3
Divide by zero error encountered.
The statement has been terminated.

(1 row(s) affected)
```

This is because we would need to add error handling to capture the error and rollback the entire transaction. We introduce error handling by using the TRY/CATCH statements. We use the Begin Try and End Try statements around the code we would like to try and run. If there are not any errors the code will run successfully. If there are errors, our code will be sent immediately to the Begin Catch and End Catch block of code to be handled appropriately. Notice we have also indented the code to make it easier to see where the error handling and transaction statements end and begin. (For those familiar with .NET language, SQL does not include a FINALLY block with its error handling.)



```
Begin Try
    Begin Transaction BankUpdate
        UPDATE Accounting.BankAccounts
        SET Balance -= 2/0
        WHERE AcctID = 1

        UPDATE Accounting.BankAccounts
        SET Balance += 200
        WHERE AcctID = 2
    Commit Transaction
End Try
Begin Catch
    Rollback Transaction
    Print 'Error in code Transaction not complete.'
End Catch
```

When we TRY to run this code an error is caught and thrown into the catch section to be handled. In the CATCH section we used the Rollback Transaction statement to undo the entire transaction. The following message shows that not only did the first UPDATE statement not run; neither did the second statement in the transaction.



```
(0 row(s) affected)
Error in code Transaction not complete.
```

By using transactions we ensure that our data stays consistent. Another way of turning on transactions is by using the SET IMPLICT_TRANSACTION ON statement. This would implicitly turn on transactions as soon as the first statement runs without having to use the Begin Transaction statement. However, it will not finish until an explicit Commit Transaction or Rollback Transaction is reached. For better control and readability, it is still best practice to use Explicit Transactions with the Begin Transaction statement.

Finally, if we did not want to include the error handling but still use the functionality of the transactions. We could use the SET XACT_ABORT ON statement. This statement would terminate the transaction once the error was reached and cancel out the rest of the code. However, we would not be able to add any other error handling functionality without using the TRY/CATCH error handling blocks.