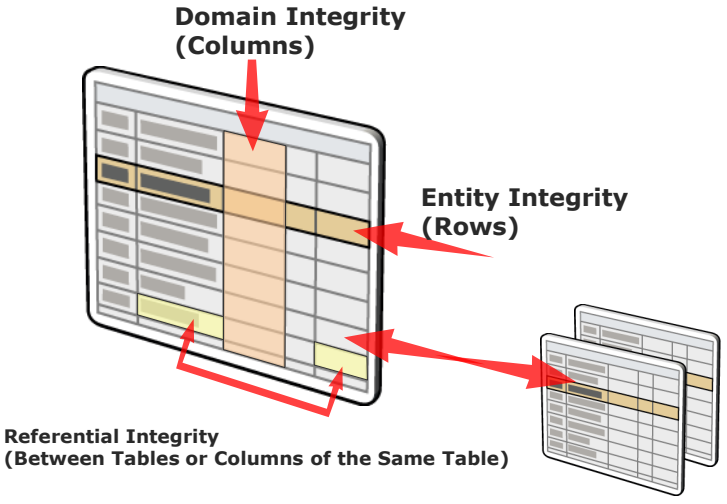
Databases I

Etelka Szendrői (PhD)
Associate professor
szendroi@mik.pte.hu

Constraints

Types of Data Integrity



What Are Constraints?

Integrity type	Constraint type	Description
Domain	DEFAULT	Specifies default value for column
	CHECK	Specifies allowed value for column
	FOREIGN KEY	Specifies column in which values must exist
	NULL	Specifies whether NULL is permitted
Entity	PRIMARY KEY	Identifies each row uniquely
	UNIQUE	Prevents duplication of nonprimary keys
Referential	FOREIGN KEY	Defines columns whose value must match the primary key of this table
	CHECK	Specifies the allowed value for a column based on the contents of another column

PRIMARY KEY constraint

A ***PRIMARY KEY*** is an important concept of designing a database table as it provides an attribute or set of attributes used to uniquely identify each row in the table

A table can only have one primary key which is created using a primary key constraint and enforced by creating a unique index on the primary key columns

A column that participates in the primary key constraint cannot accept null values

To add a PRIMARY KEY constraint to an existing table use the following command

```
ALTER TABLE Production.TransactionHistoryArchive
ADD CONSTRAINT PK_TransactionHistoryArchive_TransactionID
PRIMARY KEY CLUSTERED (TransactionID);
```

FOREIGN KEY constraint

A ***FOREIGN KEY*** is a column or combination of columns that are used to establish a link between data in two tables. The columns used to create the primary key in one table are also used to create the foreign key constraint and can be used to reference data in the same table or in another table

A foreign key does not have to reference a primary key, it can be defined to reference a unique constraint in either the same table or in another table

To add a FOREIGN KEY constraint to an existing table use the following command

```
ALTER TABLE Sales.SalesOrderHeaderSalesReason
ADD CONSTRAINT FK_SalesReason
FOREIGN KEY (SalesReasonID)
REFERENCES Sales.SalesReason (SalesReasonID)
ON DELETE CASCADE
ON UPDATE CASCADE ;
```

Cascading Referential Integrity

Controlled by CASCADE clause of the FOREIGN KEY constraint

Cascade option	UPDATE behavior	DELETE behavior
NO ACTION (Default)	Raise error; roll back operation	
CASCADE	Update foreign keys in referencing tables	Delete rows in referencing tables
SET NULL	Set foreign keys in referencing tables to NULL	
SET DEFAULT	Set foreign keys in referencing tables to DEFAULT values	

UNIQUE constraints

A ***UNIQUE constraint*** is created to ensure no duplicate values are entered in specific columns that do not participate in a primary key

Creating a UNIQUE constraint automatically creates a corresponding unique index

To create a UNIQUE constraint while creating a table use the following command

```
CREATE TABLE Production.TransactionHistoryArchive4
(TransactionID int NOT NULL,
CONSTRAINT AK_TransactionID UNIQUE(TransactionID) );
```

CHECK constraints

A ***CHECK constraint*** is created in a table to specify the data values that are acceptable in one or more columns

To create a CHECK constraint after creating a table use the following command

```
ALTER TABLE DBO.NewTable
ADD ZipCode int NULL
CONSTRAINT CHK_ZipCode
CHECK (ZipCode LIKE '[0-9][0-9][0-9][0-9][0-9]`);
```

DEFAULT constraints

A ***DEFAUT constraint*** is a special case of a column default that is applied when an INSERT statement doesn't explicitly assign a particular value. In other words, the column default is what the column will get as a value by default

To create a DEFAULT constraint on an existing table use the following command

```
ALTER TABLE Sales.CountryRegionCurrency
ADD CONSTRAINT Default_Country
DEFAULT 'USA' FOR CountryRegionCode
```

OUTPUT clause

The ***OUTPUT*** clause is used to return information from, or expressions based on, each row affected by an INSERT, UPDATE, DELETE, or MERGE statement. These results can be returned to the processing application for use in such things as confirmation messages or archiving

The following example deletes all rows in the ShoppingCartItem table. The clause OUTPUT **deleted.*** specifies that all columns in the deleted rows, be returned to the calling application which in this case was the Query Editor

```
DELETE Sales.ShoppingCartItem OUTPUT DELETED.*
WHERE ShoppingCartID = 20621;
--Verify the rows in the table matching the WHERE clause
have been deleted.
SELECT COUNT(*) AS [Rows in Table]
FROM Sales.ShoppingCartItem
WHERE ShoppingCartID = 20621;
```

T-SQL Programming

T-SQL programming

Declaring and Initializing a Variable:

Store values in the computer memory. Every variable has *type and value*

To use a variable, you must first declare it.

DECLARE @name type
example.:

DECLARE @myProdName nvarchar(40), @myProdID int

You assign a value to a variable after you declare it by using the SET statement or by using the SELECT statement.

Example:

SET @myProdName='Tea'
SET @myProdID=7

Query with parameters

We can use variables in the WHERE clause.

List those products which ProductID is the same as the @myProdID parameter value or the product name is equal with the @myProdName parameter:

```
DECLARE @myProdName nvarchar(40), @myProdID int
SET @myProdName='Pavlova'
SET @myProdID=7

SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID=@myProdID OR
    ProductName=@myProdName;
```

Results | Messages

	ProductID	ProductName	UnitPrice
1	7	Uncle Bob's Organic Dried Pears	30,00
2	16	Pavlova	17,45

Control Statements

IF ….ELSE
WHILE
GOTO
RETURN
Break/Continue
Try/Catch
WaitFor

Decision Statement - IF statement

IF condition
    [BEGIN]
    statement(s)
    [END]
ELSE
    [BEGIN]
    statement(s)
    [END]

IF structure

```
IF (SELECT COUNT(*) FROM Products WHERE UnitPrice<5)>0
BEGIN
    PRINT 'A következő termékek egységára kisebb mint 5$'
    SELECT ProductID, ProductName, UnitPrice
    FROM Products
    Where UnitPrice<5
END
ELSE
BEGIN
    PRINT 'Nincs olyan termék aminek az egységára kisebb mint 5$'
END
```

Results | Messages

	ProductID	ProductName	UnitPrice
1	24	Guaraná Fantástica	4,50
2	33	Geitost	2,50

CASE expression

You can use to apply conditional logic to determine the returned value.

The CASE expression has two forms—the *simple* form and the *searched* form.

The simple form compares an input expression to multiple possible scalar when expressions and returns the result expression associated with the first match. If there's no match and an ELSE clause is specified, the else expression is returned.

The simple form syntax:

```
CASE input expression
    WHEN expression
    THEN result expression
    ELSE result expression
END
```

Example of simple Case expression

```
SELECT CASE ProductLine
   WHEN 'R' THEN 'Road'
   WHEN 'M' THEN 'Mountain'
   WHEN 'T' THEN 'Touring'
   WHEN 'S' THEN 'Other'
   ELSE 'Parts'
END AS Category,
Name AS ProductName,
ProductNumber
FROM AdventureWorks2012.Production.Product
ORDER BY ProductName
```

Part of the result->

Results | Messages

	Category	ProductName	ProductNumber
1	Parts	Adjustable Race	AR-5381
2	Mountain	All-Purpose Bike Stand	ST-1401
3	Other	AWC Logo Cap	CA-1098
4	Parts	BB Ball Bearing	BE-2349
5	Parts	Bearing Ball	BA-8327
6	Other	Bike Wash - Dissolver	CL-9009
7	Parts	Blade	BL-2036
8	Other	Cable Lock	LO-C100
9	Parts	Chain	CH-0234
10	Parts	Chain Stays	CS-2812
11	Parts	Chainring	CR-7833

CASE Expression

The *searched* form of the CASE expression is more flexible. Instead of comparing an input expression to multiple possible expressions, it uses predicates in the WHEN clauses, and the first predicate that evaluates to true determines which *WHEN* expression is returned. If none is true, the CASE expression returns the *ELSE* expression.

```
SELECT ProductID, ProductName, UnitPrice,
CASE
WHEN UnitPrice < 20.00 THEN 'Low'
WHEN UnitPrice < 40.00 THEN 'Medium'
WHEN UnitPrice >= 40.00 THEN 'High'
ELSE 'Unknown'
END AS Pricerange
FROM Products;
```

Results | Messages

	ProductID	ProductName	UnitPrice	Pricerange
1	1	Chai	18,00	Low
2	2	Chang	19,00	Low
3	3	Aniseed Syrup	10,00	Low
4	4	Chef Anton's Cajun Seasoning	22,00	Medium
5	5	Chef Anton's Gumbo Mix	21,35	Medium
6	6	Grandma's Boysenberry Spread	25,00	Medium
7	7	Uncle Bob's Organic Dried Pears	30,00	Medium
8	8	Northwoods Cranberry Sauce	40,00	High
9	9	Mishi Kobe Niku	97,00	High
10	10	Ikura	31,00	Medium
11	11	Queso Cabrales	21,00	Medium
12	12	Queso Manchego La Pastora	38,00	Medium
13	13	Konbu	6,00	Low
14	14	Tofu	23,25	Medium
15	15	Genen Shouyu	15,50	Low
16	16	Pavlova	17,45	Low
17	17	Alice Mutton	39,00	Medium

Here another example of simple Case expression and variables

```
DECLARE @State nchar(2), @StateName nvarchar(15)
SET @State='MA'
SET @StateName=CASE @State
WHEN 'CA' THEN 'California'
WHEN 'MA' THEN 'Massachusetts'
WHEN 'NY' THEN 'New York'
END
SELECT @StateName as StateName;
```

Results | Messages

StateName
Massachusetts

Simple Case:
Check the last digit of OrderId in Orders table and show the value as a text, but only if it is in 1-5 range. Otherwise the result let be the text 'other.' List the first 15 rows from the table.

```
SELECT TOP 15 OrderID, OrderDate, OrderID%10 as LastDigit,
    CASE OrderID%10
        WHEN 1 THEN 'One'
        WHEN 2 THEN 'Two'
        WHEN 3 THEN 'Three'
        WHEN 4 THEN 'Four'
        WHEN 5 THEN 'Five'
        ELSE 'Other'
    END as LastDigitText
FROM Orders;
```

100 % | Results | Messages

	OrderID	OrderDate	LastDigit	LastDigitText
1	10249	1996-07-05 00:00:00.000	9	Other
2	10250	1996-07-08 00:00:00.000	0	Other
3	10251	1996-07-08 00:00:00.000	1	One
4	10252	1996-07-09 00:00:00.000	2	Two
5	10253	1996-07-10 00:00:00.000	3	Three
6	10254	1996-07-11 00:00:00.000	4	Four
7	10255	1996-07-12 00:00:00.000	5	Five
8	10256	1996-07-15 00:00:00.000	6	Other
9	10257	1996-07-16 00:00:00.000	7	Other
10	10258	1996-07-17 00:00:00.000	8	Other
11	10259	1996-07-18 00:00:00.000	9	Other
12	10260	1996-07-19 00:00:00.000	0	Other
13	10261	1996-07-19 00:00:00.000	1	One
14	10262	1996-07-22 00:00:00.000	2	Two
15	10263	1996-07-23 00:00:00.000	3	Three

Performing conditional tests with IIF

IIF returns one of two values, depending on a logical test
Shorthand for a two-outcome CASE expression

IIF Element	Comments
Boolean_expression	Logical test evaluating to TRUE, FALSE, or UNKNOWN
True_value	Value returned if expression evaluates to TRUE
False_value	Value returned if expression evaluates to FALSE or UNKNOWN

```
SELECT ProductID, ListPrice,
IIF(ListPrice > 50, 'high', 'low') AS PricePoint
FROM Production.Product;
```

Selecting items from a list with CHOOSE

CHOOSE returns an item from a list as specified by an index value

CHOOSE Element	Comments
Index	Integer that represents position in list
Value_list	List of values of any data type to be returned

CHOOSE example:

```
SELECT CHOOSE (3, 'Beverages', 'Condiments', 'Confections') AS
choose_result;
```

```
choose_result
-------------
Confections
```

LOOPS - WHILE

With the WHILE construct, you can create loops inside T-SQL in order to execute a statement block as long as a condition continues to evaluate to true.

Syntax:

WHILE condition
 [Begin]
  statements
[End]

Example:

```
DECLARE @Count int
SET @Count=5
WHILE @Count>0
BEGIN
  PRINT 'counter= '+CONVERT(nvarchar(3),@Count)
  SET @Count=@Count-1
  END;
```

```
00 %
Messages
counter= 5
counter= 4
counter= 3
counter= 2
counter= 1
```

CONVERT

If you want to print numeric values, you have to convert them to text type, as you can see in the earlier example. You can use the CONVERT() or the CAST() functions.

Example:

```
PRINT 'counter= '+CONVERT(nvarchar(3),@Count)
```

OR:

```
PRINT 'counter= '+CAST(@Count AS nvarchar)
```

CONTINUE and BREAK statements

Inside the WHILE loop, you can use a BREAK statement to end the loop immediately

and a

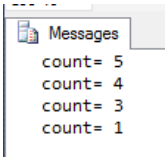
CONTINUE statement to cause execution to jump back to the beginning of the loop.

CONTINUE

IF @count =2, jump back to the beginning of the loop

```
DECLARE @count int
 SET @count=5
WHILE (@count>0)
BEGIN
     PRINT 'count= '+ CONVERT(nvarchar, @count)
     SET @count=@count-1
     IF(@count=2)
          BEGIN
           SET @count=@count-1
           CONTINUE
          END
END
```

The result:

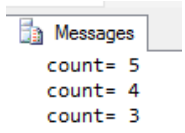


33

BREAK Statement

IF @Count=2 BREAK statement ends the loop immediately

```
DECLARE @count int
SET @count=5
WHILE (@count>0)
BEGIN
    PRINT 'count= '+ CONVERT(nvarchar, @count)
    SET @count=@count-1
    IF(@count=2)
        BEGIN
          BREAK
        END
END
```



WAITFOR statement

The WAITFOR command can cause execution of statements to pause for a specified period of time. WAITFOR has three options: WAITFOR DELAY, WAITFOR TIME, and WAITFOR RECEIVE.
(WAITFOR RECEIVE is used only with Service Broker.)

WAITFOR DELAY causes the execution to delay for a requested duration.

For example, the following WAITFOR DELAY pauses code execution for 5 seconds.

`WAITFOR DELAY '00:00:05'` – pauses 5 seconds

WAITFOR TIME, on the other hand, pauses execution to wait for a specific time.

For example, the following code waits until '20:15:10'.

```
WAITFOR TIME '20:15:10'
```

RETURN statement

Whenever a RETURN is executed, execution of the stored procedure or function ends and control returns to the caller.

You can use more than one RETURN command in a procedure.

RETURN by itself causes SQL Server to send a status code back to the caller. The statuses are 0 for successful and a negative number if there is an error.

You can send your own return codes back to the caller by inserting an integer value after the RETURN statement.

Syntax:

```
RETURN [int-expression]
```

Using cursors

SQL Server is built to process sets of data. However, there are times when you need to process data one row at a time.

The result of a SELECT statement is returned to a server-side object called *cursor*, which allows you to access one row at a time within the result set and even allows scrolling forward as well as backward through the result set.

Cursors

Usage steps:

Declare variables to store fields from SELECT statement

Declare cursor and define the SELECT statement

Open the cursor

Fetch rows from the cursor

Close the cursor

Usage of Cursors

1. *Declare variables* to store the retrieved fields from the SELECT statement
2. *Declare cursor*: is used to define the SELECT statement that is the basis for the rows in the cursor.
3. *Open cursor*: Open causes the SELECT statement to be executed and loads the rows into a memory structure.
4. *FETCH* is used to retrieve one row at a time from the cursor.
5. *CLOSE* is used to close the processing on the cursor. *DEALLOCATE* is used to remove the cursor and release the memory structures containing the cursor result set.

An example: Using cursor list the Products table rows

```
-- Using Cursor
-- 1. declare variables
DECLARE @myProductID int, @myProductName nvarchar(40), @myUnitprice money
-- 2. declare cursor
DECLARE ProductCursor CURSOR FOR
SELECT ProductID, ProductName, UnitPrice
FROM Products
WHERE ProductID<=10
-- 3. Open the cursor
OPEN ProductCursor
```

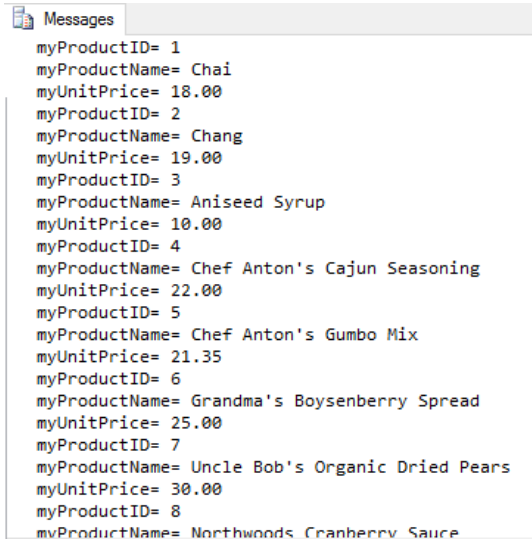
Example (cont.)

```
-- 4. Fetch rows
FETCH NEXT FROM ProductCursor INTO @myProductID, @myProductName, @myUnitPrice
WHILE @@FETCH_STATUS=0
BEGIN
   PRINT 'myProductID= '+CAST(@myProductID AS nvarchar)
   PRINT 'myProductName= '+@myProductName
   PRINT 'myUnitPrice= '+CAST(@myUnitPrice AS nvarchar)
   FETCH NEXT FROM ProductCursor INTO @myProductID, @myProductName, @myUnitPrice
END
-- 5. Close the cursor
CLOSE ProductCursor
DEALLOCATE ProductCursor
```

0 – when the previous fetch was successful
-1 – when the row is beyond the result set
-2 – when the row fetched is missing

close the cursor and the DEALLOCATE it.

The Result



User Defined Functions

The purpose of a user-defined function (UDF) is to encapsulate reusable T-SQL code and return a scalar value or a table to the caller.

Types:

Scalar functions: retrieve only one value

Inline table-valued function: return a table; contains single line of code

Table-valued function: return a table; multiple lines of code is called a multistatement table-valued UDF.

User-defined Function

Create:
CREATE FUNCTION funcName(parameters)
Modify:
ALTER FUNCTION funcName
Delete:
DROP FUNCTION funcName
Calling:
schemaname.functionName
example: dbo.DiscountPrice()

User Defined Function

Example: Take an OriginalPrice and a discount value and return the result of multiplying them together.

```
CREATE FUNCTION DiscountPrice(@OriginalPrice money, @Discount float)
RETURNS money
AS
BEGIN
  RETURN @OriginalPrice * @Discount
END
```

Execute:

```
/* Futtatás */
SELECT dbo.DiscountPrice(120,0.2)
```

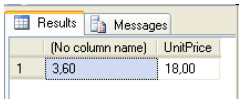
Results | Messages

	(No column name)
1	24,00

User defined Function

The input parameter can be a column value from a table:

```
SELECT dbo.DiscountPrice(UnitPrice,0.2), UnitPrice
FROM Products
WHERE ProductID=1;
```



- The parameter can be a local variable

```
/* Futtatás */
DECLARE @Engedmeny float
SET @Engedmeny=0.2
SELECT dbo.DiscountPrice(UnitPrice, @Engedmeny), UnitPrice
FROM Products
WHERE ProductID=1;
```

Inline table-valued function

A table-valued UDF returns a table rather than a single value to the caller. As a result, it can be called in a T-SQL query wherever a table is expected, which is in the FROM clause.

An **inline** table-valued function is the only type of UDF that can be written without a BEGIN/END block.

An inline table-valued UDF contains a single SELECT statement that returns a table.

Example: Create a function to retrieve the rows from Products table where the UnitsInStock value less or equel with the input parameter (@ReorderLevel)

```
CREATE FUNCTION ProductsToBeReordered(@ReorderLevel int)
RETURNS table
AS
RETURN
(
  SELECT *
  FROM Products
  WHERE UnitsInStock <= @ReorderLevel
)
```

To call the function, embed it in the FROM clause of a SELECT statement, but be sure to supply the required parameters.

```
SELECT *
FROM ProductsToBeReordered(6);
```

Results | Messages

	ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder	ReorderLevel
1	5	Chef Anton's Gumbo Mix	2	2	36 boxes	21,35	0	0	0
2	8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40,00	6	0	0
3	17	Alice Mutton	7	6	20 - 1 kg tins	39,00	0	0	0
4	21	Sir Rodney's Scones	8	3	24 pkgs. x 4 pieces	10,00	3	40	5
5	29	Thüringer Rostbratwurst	12	6	50 bags x 30 sausgs.	123,79	0	0	0
6	31	Gorgonzola Telino	14	4	12 - 100 g pkgs	12,50	0	70	20
7	45	Rogede sild	21	8	1k pkg.	9,50	5	70	15
8	53	Perth Pasties	24	6	48 pieces	32,80	0	0	0
9	66	Louisiana Hot Spiced Okra	2	2	24 - 8 oz jars	17,00	4	100	20
10	68	Scottish Longbreads	8	3	10 boxes x 8 pieces	12,50	6	10	15
11	74	Longlife Tofu	4	7	5 kg pkg.	10,00	4	20	5

We can filter the result retrieved from the function

```
SELECT ProductID, ProductName, UnitsInStock
FROM ProductsToBeReordered(10)
WHERE ProductID <=50;
```

Results | Messages

	ProductID	ProductName	UnitsInStock
1	5	Chef Anton's Gumbo Mix	0
2	8	Northwoods Cranberry Sauce	6
3	17	Alice Mutton	0
4	21	Sir Rodney's Scones	3
5	29	Thüringer Rostbratwurst	0
6	30	Nord-Ost Matjeshering	10
7	31	Gorgonzola Telino	0
8	32	Mascarpone Fabioli	9
9	45	Rogede sild	5
10	49	Maxilaku	10

Multistatement Table-Valued UDF

Multistatement table-valued UDF can consist of multiple lines of T-SQL code.

A multistatement table-valued UDF has a RETURN statement at the end of the function body.

Solve the earlier problem with multistatement table-valued function! The returntable will contain the **ProductID**, **ProductName** and **UnitsInStock** fields and extends the fields with a new one the **Reorder** column which contains **yes/no** values according to the result of comparing UnitsInStock and @ReorderLevel values.

```
CREATE FUNCTION ProductsToBeReordered2(@ReorderLevel int)
RETURNS @MyProducts table
(
  ProductID int,
  ProductName nvarchar(40),
  UnitsInStock smallint,
  Reorder nvarchar(3)
)
AS
BEGIN

  INSERT INTO @MyProducts
    SELECT ProductID, ProductName, UnitsInStock, 'No'
    FROM Products;

  UPDATE @MyProducts
  SET Reorder = 'Yes'
  WHERE UnitsInStock <= @ReorderLevel

  RETURN

END
```

You must define the table to be returned as a table variable and insert data into the table variable. The RETURN statement just ends the function and is not used to send any data back to the caller.

The result of the execution (only the first 12 rows):

```
SELECT *
FROM ProductsToBeReordered2(20);
```

Results | Messages

	ProductID	ProductName	UnitsInStock	Reorder
1	1	Chai	39	No
2	2	Chang	17	Yes
3	3	Aniseed Syrup	13	Yes
4	4	Chef Anton's Cajun Seasoning	53	No
5	5	Chef Anton's Gumbo Mix	0	Yes
6	6	Grandma's Boysenberry Spread	120	No
7	7	Uncle Bob's Organic Dried Pears	15	Yes
8	8	Northwoods Cranberry Sauce	6	Yes
9	9	Mishi Kobe Niku	29	No
10	10	Ikura	31	No
11	11	Queso Cabrales	22	No
12	12	Queso Manchego La Pastora	86	No