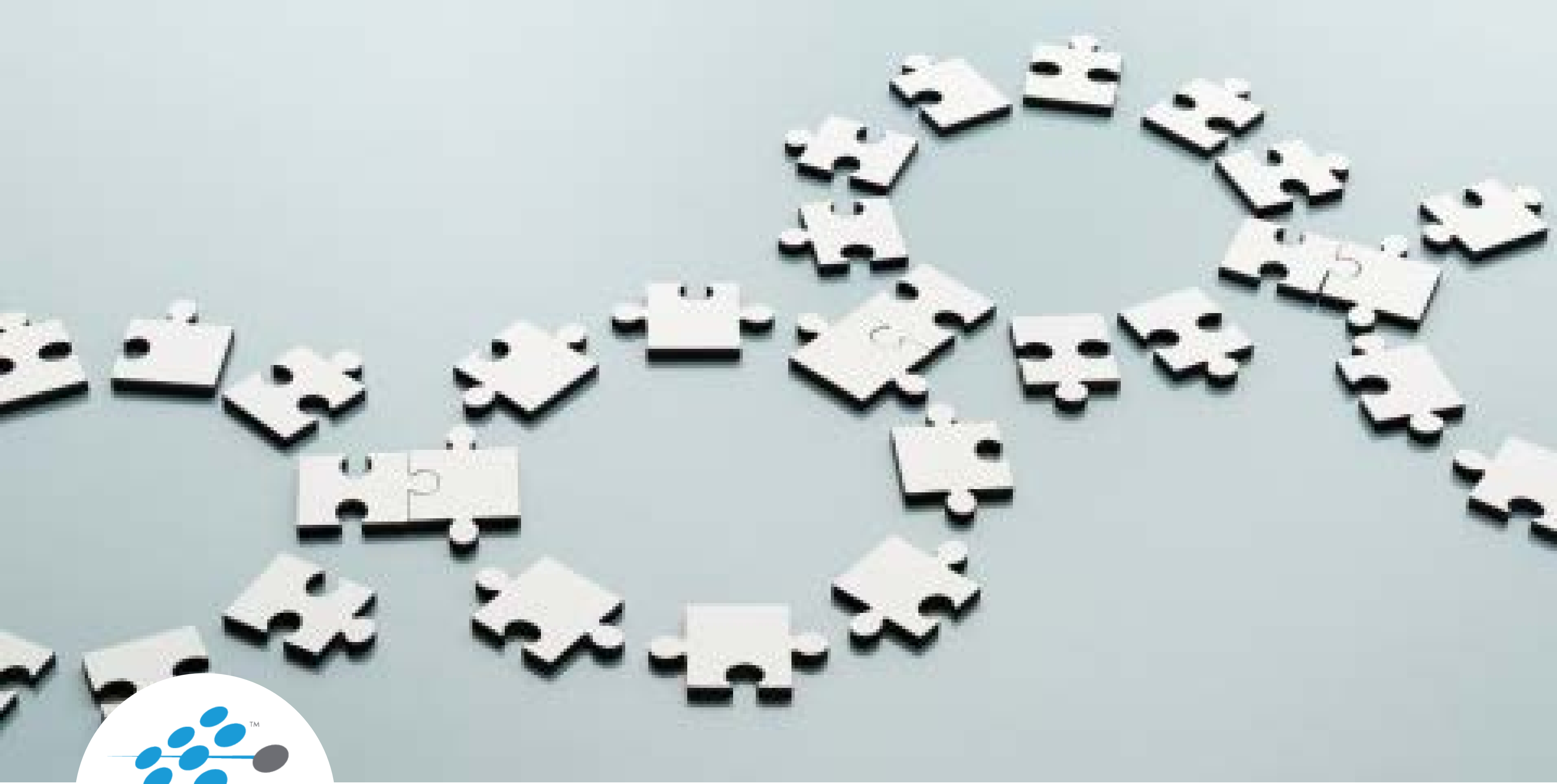


HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 1

ECL Online Training: Overview – Format and Prerequisites

LexisNexis®

Risk Solutions

Welcome! – Course Overview

Welcome to the HPCC Systems Online ECL Training Series!

We have designed these lessons to be specialized and timed for convenient learning sessions that fit your schedule.

Lessons are mixed with slides and online demonstration using the ECL IDE and/or Eclipse.

Some lessons will have lab exercises or interactive questions to help you reinforce what you have learned.

Meet Your Instructors

Richard Taylor

Chief Trainer at LexisNexis for over 14 years

ECL Guru and member of Documentation team

Bob Foreman

Senior Trainer at LexisNexis for over 4 years

20+ years of experience as Technical Trainer

Before we get started…

First, you will need to have **access to any HPCC cluster**. We recommend using an HPCC VM for these lessons, but if you have access to an actual cluster, you can use that.

Download the HPCC VM at the HPCC web site:

HPCC Virtual Machine (including a link to download the VM Player):
http://hpccsystems.com/download/hpcc-vm-image

Here is the latest PDF to help you get started with downloading and running the HPCC VM for the first time:

http://hpccsystems.com/download/docs/running-hpcc-vm

The links to the ECL IDE and all associated documentation is located in the ECL Watch tool located in your VM. We'll show you how to get there.

Before we get started…

As the PDF on the last slide describes, to run the latest VM version of the HPCC System, you will need the VM player (version 3.0 or later) from VMWare®.

Before proceeding to the next slide, you will need to perform the following as described in the PDF:

Download and install the VM Player

Download the HPCC virtual machine image from HPCC Systems.

Open the image in the VM Player

Write down the IP address that is displayed in the VM Player, and proceed to the next slide to install the ECL IDE and associated documentation.

Before we get started

To install the latest IDE and documentation associated with your HPCC VM, perform the following steps:

Open your favorite browser, and go to the **ECL Watch tool** by entering **http://nnn.nnn.nnn.nnn:8010**, where *nnn.nnn.nnn.nnn* is your Virtual Machine's IP address.

From the ECL Watch page, click on the **Resources > Browse** link in the menu on the left side.

In the list shown, click on the **ECL IDE Installer** link near the bottom of this page. When prompted save this file to your PC and then run it—do not run directly from your browser. This is a Windows installer for a Windows application.

Before we get started

You will also need to download the training data that we will use in this course at the designated location where you launched this lesson.

After downloading the file, extract and save it to a location on your local machine that you will remember. In a later lesson we will transfer this file to your HPCC VM landing zone.

Lesson Sequence

- ✓ HPCC Overview
- ✓ Introduction to the ECL IDE and ECL Watch
- ✓ Spraying Data
- ✓ ECL Language Basics Overview
- ✓ Basic Query and Filter Exercises
- ✓ Creating ECL Definitions (hands-on)
- ✓ More ECL Functions
- ✓ More Exercises
- ✓ Transformation using SORT and DEDUP

What are the primary goals in this series?

- ✓ Knowing HPCC Architecture and Terminology
- ✓ Familiarity with ECL (Enterprise Control Language) Programming tools
  - ✓ ECL IDE (or Eclipse IDE)
  - ✓ ECL Watch
- ✓ Understanding ECL Concepts and Syntax
- ✓ Mastery of five ECL fundamentals:
  - ✓ Creating and recognizing the following ECL Definition Types:

✓ Boolean	- TRUE/FALSE expressions
✓ Value	- Scalar value expressions
✓ Set	- Sets of scalar values
✓ Record Set	- Sets of data records

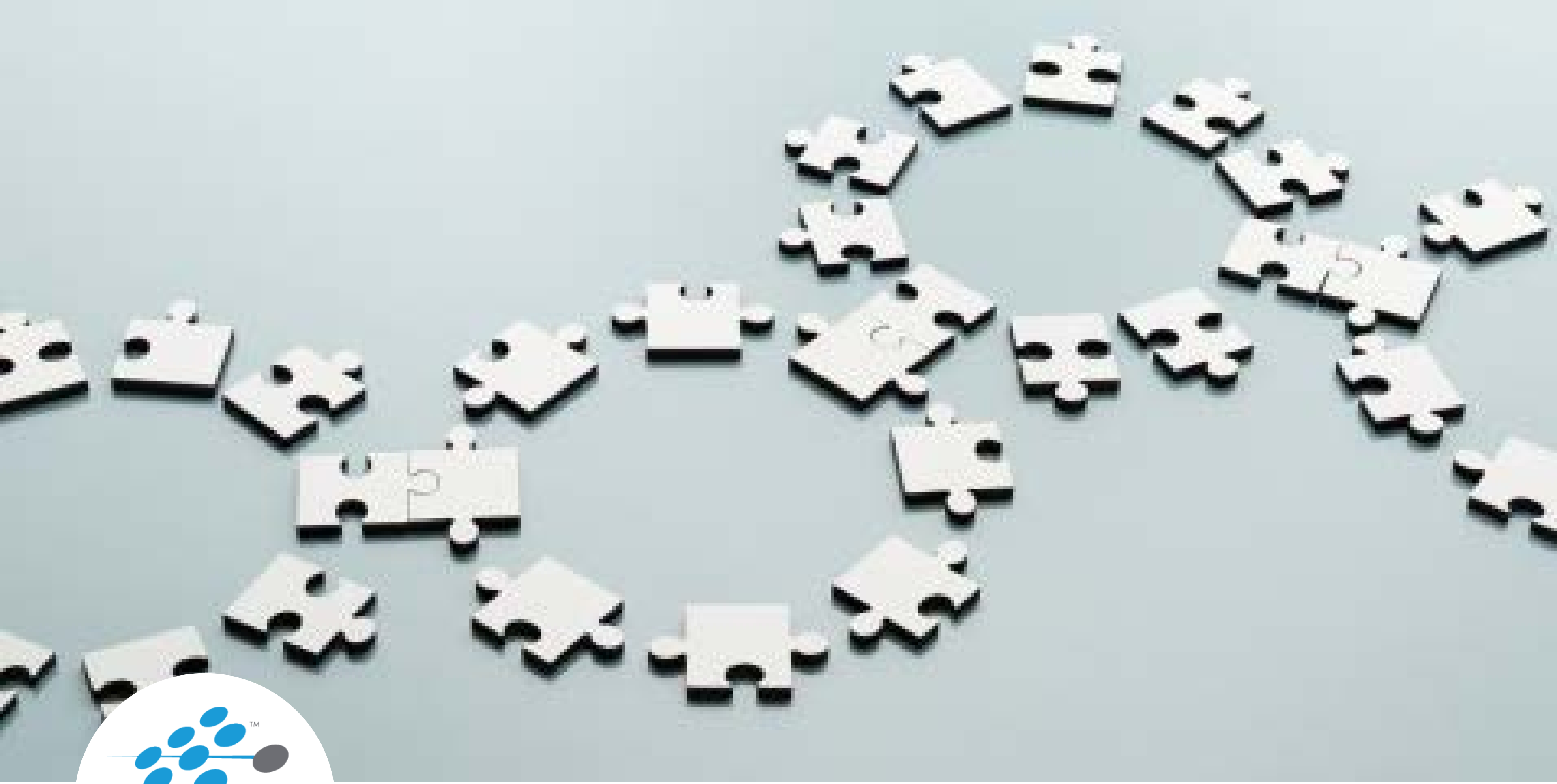
  - ✓ Record Set Filtering - "WHERE" clauses

End of this lesson

This concludes this lesson.

Next lesson:

Introduction to HPCC – Architectural Overview



HPCC SYSTEMS®

HPCC Systems:
Introduction to the Enterprise Control Language

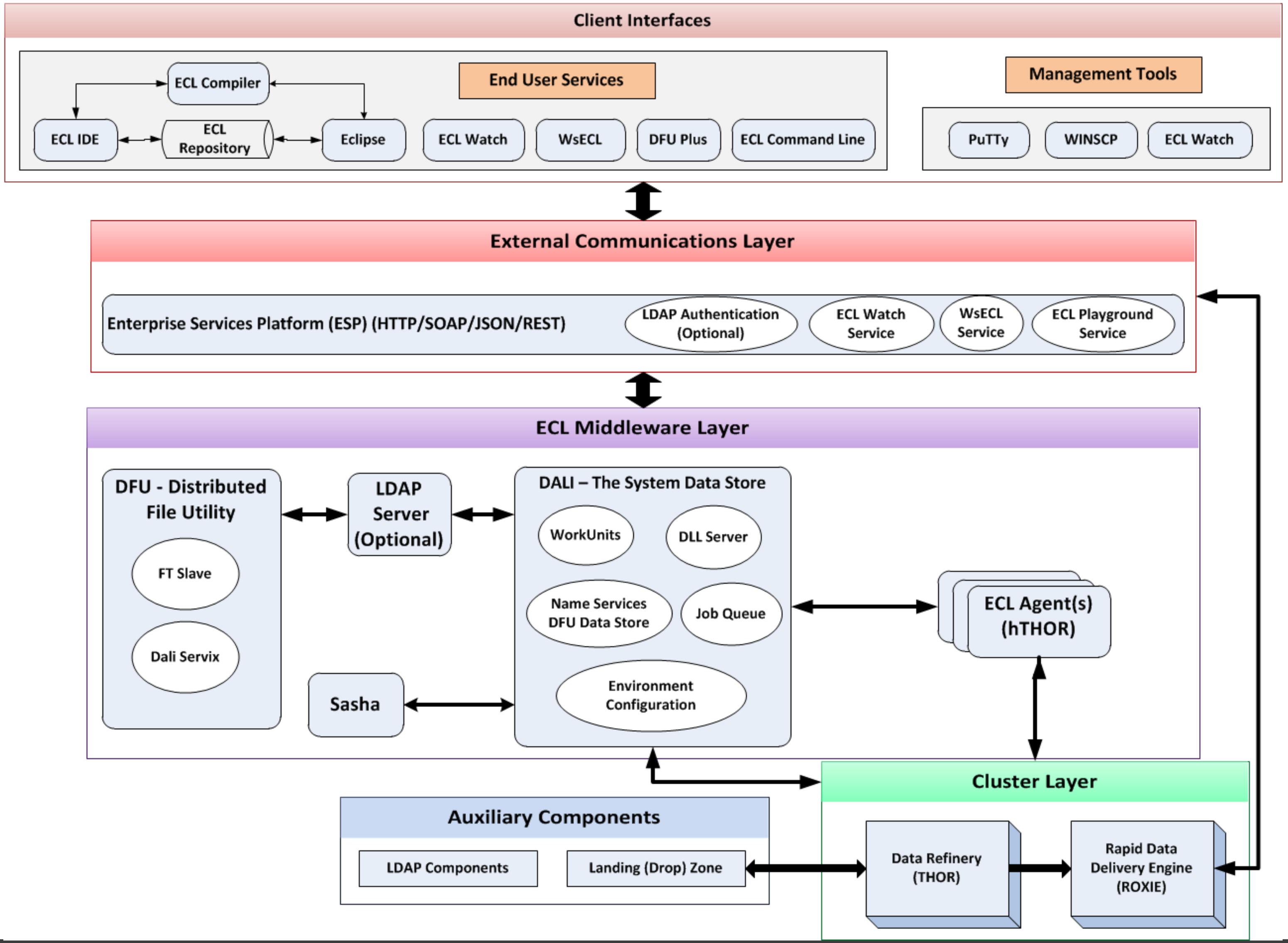
Lesson 2

Introduction to HPCC – An Architectural Overview

LexisNexis®

Risk Solutions

HPCC Architecture



Components of the HPCC

Clusters:
- THOR
- HTHOR (ECL Agent)
- ROXIE

System servers:
- Dali
- Sasha
- DFU Server
- ECLCC Server

More System servers:
- ECL Agent
- ESP Server
- LDAP

Client interfaces:
- ECL IDE
- ECL Watch
- Command line tools

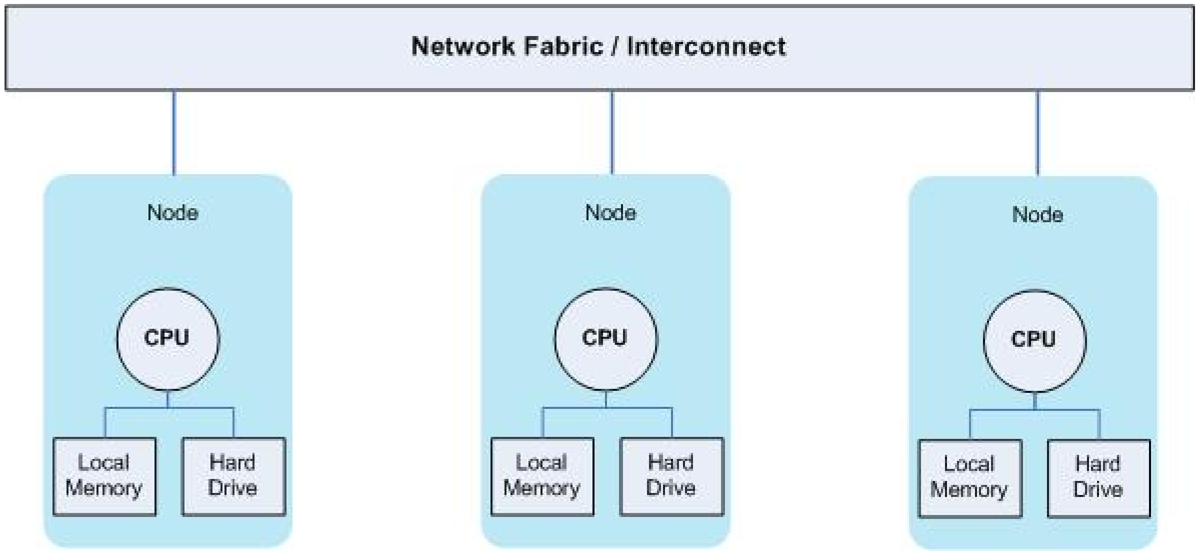
HPCC Clusters

LN’s HPCC environment contains clusters which you define and use according to your needs.

There are two types of clusters in the HPCC:

- Data Refinery (THOR) – Used to process every one of billions of records in order to create billions of "improved" records.
  - ECL Agent (hThor) is also used to process simple jobs that would be an inefficient use of the THOR cluster.
- Rapid Data Delivery Engine (ROXIE) – Used to search quickly for a particular record or set of records.

Hardware



Cluster of commercial off-the-shelf components (COTS). Ideally, components are homogeneous (all processing/disk storage components same, but that is not required) and the system is tightly coupled. Although similar to a Grid system, nodes are managed *en masse* instead of individually allowing coordinated processing like global sorts.

Data on the HPCC

THOR/ECL Agent (HTHOR)

- Data loading is controlled through the Distributed File Utility (DFU) Server.
- Data typically arrives on the Landing Zone (for example, by FTP).
- File movement is initiated by DFU.
- Data is copied from the Landing Zone and is distributed (sprayed) into the Data Refinery (THOR).
- Data can now be further processed (Extract Transform and Load process).
- Data retrieval process places the file back on the landing zone (despraying).
- A single physical file is distributed into multiple physical files across the nodes of a cluster. The aggregate of the physical files creates one logical file that is addressed by the programmers of ECL code.

ROXIE

- Data and queries are compiled in ECL IDE and deployed (published) in ECL Watch.
- Data moves in parallel from THOR nodes to the receiving ROXIE nodes. Parallel bandwidth utilization improves the speed of putting new data into play.

The Big Picture

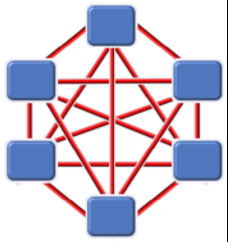
HPCC SYSTEMS

Raw Input Data

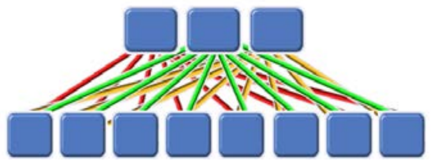
Many thousands of data files
Structured & unstructured data
Daily / weekly / monthly updates

THOR
(LN's HPCC Data Refinery)

Many hundreds of nodes
Petabytes of storage
TBs of memory



Parallel Copy Pipeline
1 GB per second (i.e. TB in 17 min)



ROXIE (LN's HPCC Rapid Data Delivery Engine)

Few hundred nodes
Hundred TBs of storage
Hundred GBs of memory

Users Search Data

Users

Thousands of users
Web / SOAP / Batch
Sub-second response times

HPCC Middleware

Dali is also known as the System Data Store.

It manages:

- Workunits.
- Logical file directory.
- Shared object services.

It is also used to:

- Configure the environment.
- Maintain the message queues that drive job execution and scheduling.
- Enforce the LDAP security restrictions for data file scopes and workunit scopes.

HPCC Middleware

The Sasha server is a companion “housekeeping” server to the Dali server.

It works independently of all other components:

- Allowing a restart to happen at any time without interfering with jobs in flight.

Sasha’s main function is to reduce the stress on the Dali server whenever possible.

Sasha:

- Archives workunits (including DFU Workunits) which:
  - Are stored in a series of folders.
  - Can be restored.
  - Can be manually moved to an alternate or off-site location.
- Reduces the resource utilization on Dali.
- Removes cached workunits and DFU recovery files.

HPCC Middleware

HPCC SYSTEMS

DFU server controls the spray and despray operations used to move data to and from THOR.

DFU services are available from:

- ECL IDE, using service libraries in ECL code.
- ECL Watch.
- DFU command line interface (DFUplus).

HPCC Middleware

ECLCC Server is the code generator and compiler that translates ECL code.

When workunits are submitted for execution on THOR, they are first converted to executable code by ECLCC Server.

For ROXIE, this process is done at deploy time so that a query can be compiled once, but then executed multiple times.

ECLCC Server also:

- Is used when the ECL IDE requests a syntax check.
- Uses a queue in order to convert workunits one at a time.
  - However, there may be multiple ECLCC Servers deployed in the system to increase throughput and they will automatically load-balance as required.

HPCC Middleware

ECL Agent (HTHOR) is a single node process for executing simple ECL Queries.

Using ECL Agent for simple queries:

- Simplifies the process.
- May mean they run faster.

There are **two** circumstances where it may be used:

- You may choose to use it as your target:
  - If you know your query will run at an acceptable level of performance on a single node.
  - If THOR is unavailable – but your query must still be simple enough to run on ECL Agent.
- The job is automatically sent:
  - If the compiler determines that a query is simple enough to be run on ECL Agent.
  - Queries that are not simple enough for ECL Agent are sent to an available THOR cluster.

HPCC Middleware

ESP Server (Enterprise Service Platform) is the inter-component communication server.

ESP Server is a framework that allows multiple services to be “plugged in” to provide various types of functionality to client applications via multiple protocols.

Examples of services that are plugged into ESP include:

- myWs_ECL
  Provides an interface to deployed queries on THOR and ROXIE cluster.
- ECL Watch
  A web-based query execution, monitoring and file management interface. It can be accessed via ECL IDE or a web browser.

Examples of protocols supported by the ESP Server framework include:

- HTTP
- SOAP
- JSON
- REST

HPCC Middleware

LDAP – Lightweight Directory Access Protocol

Works with Dali to enforce the security restrictions for data file and workunit scopes.

Genesis

A Linux server used to host the operating system for all nodes in the HPCC cluster.

HPCC Client Tools

HPCC SYSTEMS®

ECL IDE

A full-featured GUI for ECL development providing access to the ECL repository and many of the ECL Watch capabilities.

Uses various ESP services via SOAP.

Provides the easiest way to create:

- Queries into your data.
- ECL Definitions to build your queries which:
  - Are created by coding an expression that defines how some calculation or record set derivation is to be done.
  - Once defined, can be used in succeeding ECL definitions.

HPCC Client Tools

ECL Watch

A web-based query execution, monitoring and file management interface. It can be accessed via ECL IDE or a web browser.

ECL Watch allows you to:

- See information about active workunits.
- Monitor cluster activity.
- Browse through previously submitted WUs:
  - See a visual representation of the data flow within the WU.
  - Complete with statistics which are updated as the job progresses.
- Search through files and see information including:
  - Record counts and layouts.
  - Sample records.
  - The status of all system servers whether they are in clusters or not.
- View log files.
- Start and stop processes.

HPCC Client Tools

Command line tools:

- DFU PLUS
- ECL

Provide more convenient access to functionality provided by the ECL Watch web pages.

They work by communicating with the corresponding ESP service via SOAP.

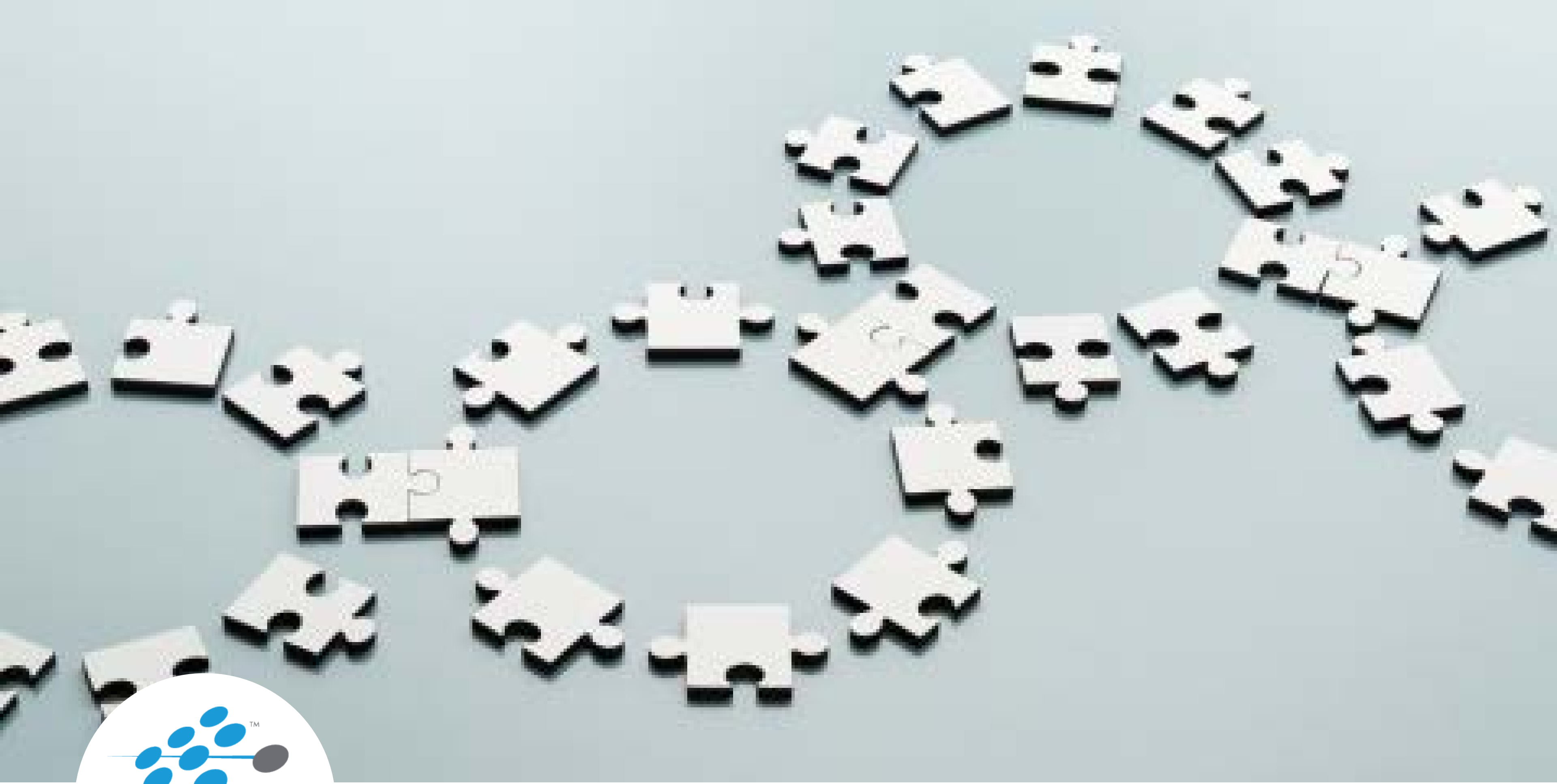
Customers can write their own client tools using the SOAP interface, where a SOAP interface has been published.

End of this lesson

This concludes this lesson.

Next lesson:

Introduction to the ECL IDE



HPCC SYSTEMS®

HPCC Systems:

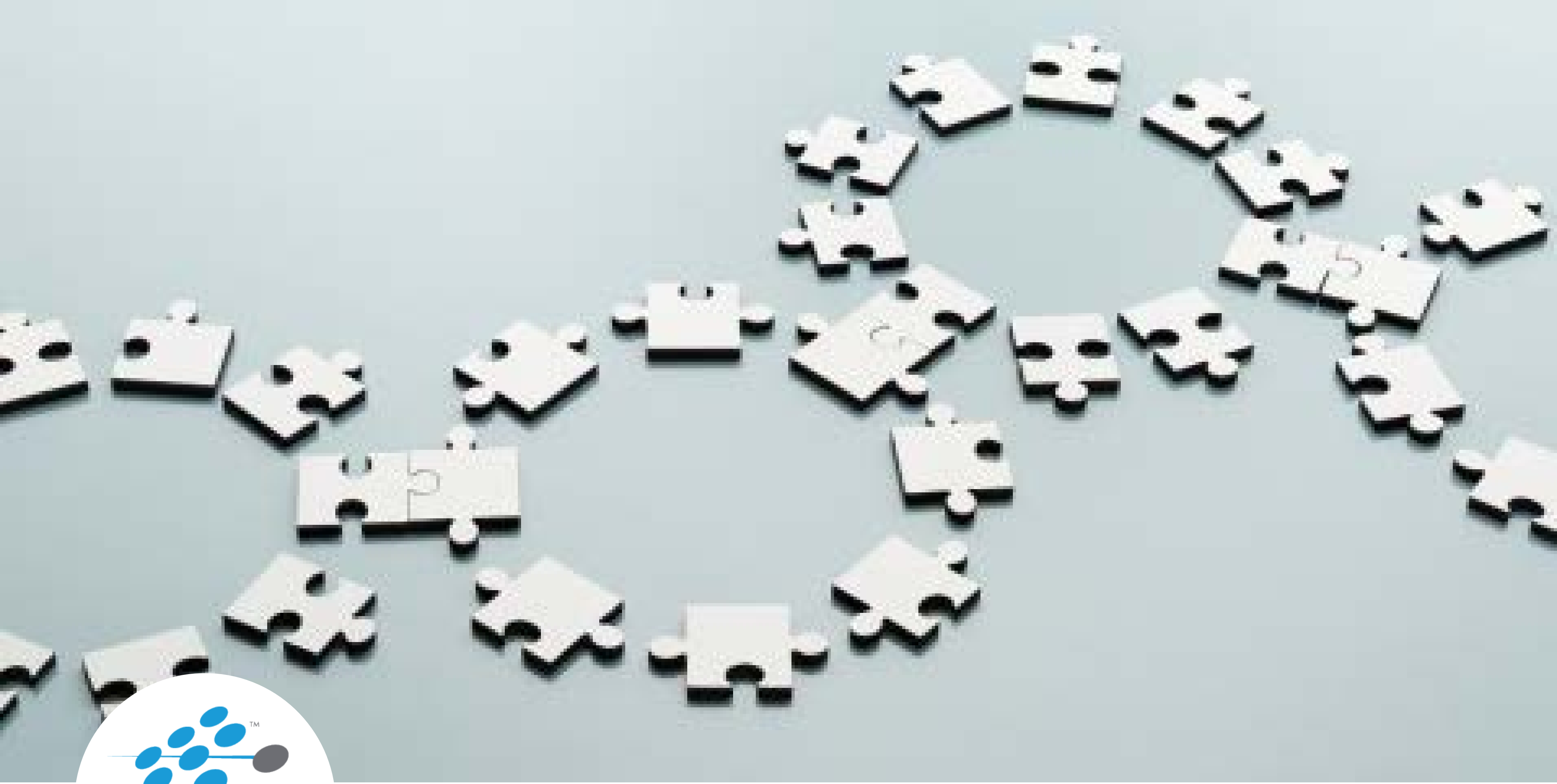
Introduction to the Enterprise Control Language

Lesson 4

Introduction to the ECL Watch – A Guided Tour – Video Demonstration

LexisNexis®

Risk Solutions



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 5

Lab Exercise 2 – Introduction to Spraying

LexisNexis®

Risk Solutions

Spraying Data Files

1. Fixed-length
2. Delimited
3. XML
4. Variable
5. BLOB

Fixed Spray Key Options

Target

– **Group:**	Select the name of THOR cluster to spray to.
– **Name Prefix:**	Part of the logical name that chosen for the file(s).
– **Target Name:**	The specific name appended to the Name Prefix.
– **Record Length:**	The size of each record.

Options

– **Overwrite:**	Check to overwrite files of the same name.
– **Compress:**	Check to compress the data.
– **Fail If No Source File**	Check to Fail Spray if Source File missing (N/A)
– **Record Structure Present**	N/A for Fixed

Delimited Spray Options:

Target

– **Group:**	Select the name of THOR cluster to spray to.
– **Name Prefix:**	Part of the logical name that chosen for the file(s).
– **Target Name:**	The specific name appended to the Name Prefix.

Options

– **Format:**	Select the file format from the drop-list.
– **Max Record Length:**	The length of longest record in the file.
– **Separator:**	The field delimiter in the source file.
– **Omit Separator:**	Check to ignore all separators
– **Escape**	The Escape code used to detect special characters
– **Line Terminator:**	The record delimiter in the source file.
– **Quote:**	The quote character in the source file.
– **Overwrite:**	Check to overwrite files of the same name.
– **Compress:**	Check to compress the data
– **Fail If No Source File**	Check to Fail Spray if Source File missing
– **Record Structure Present**	Check for record field names in first row

XML Spray Options:

Target

– **Group:**	Select the name of THOR cluster to spray to.
– **Name Prefix:**	Part of the logical name that chosen for the file(s).
– **Target Name:**	The specific name appended to the Name Prefix.
– **Row Tag:**	The XML record delimiter tag in the source file.

Options

– **Format:**	Select the file format from the drop-list.
– **Max Record Length:**	The length of longest record in the file.
– **Overwrite:**	Check to overwrite files of the same name.
– **Compress:**	Check to compress the data
– **Fail If No Source File**	Check to Fail Spray if Source File missing
– **Record Structure Present**	Check for record field names in first row

Variable Spray Options:

Target

– **Group:**	Select the name of THOR cluster to spray to.
– **Name Prefix:**	Part of the logical name that chosen for the file(s).
– **Target Name:**	The specific name appended to the Name Prefix.

Options

- **Source Type** Select the appropriate source IBM format used to spray:
  - ***Recfmv* is for IBM RDW files.** ***Recfmb* is for IBM BDW/RDW files.**
  - ***Variable*** ***Variable-Big Endian***

– **Overwrite:**	Check to overwrite files of the same name.
– **Compress:**	Check to compress the data
– **Fail If No Source File**	Check to Fail Spray if Source File missing
– **Record Structure Present**	Check for record field names in first row

BLOB Spray Options:

Target

– **Group:**	Select the name of THOR cluster to spray to.
– **Target Name:**	The specific name appended to the Name Prefix.
– **Source Path (Wildcard Enabled):**	Part of the logical name that chosen for the file(s).

Options

– **BLOB Prefix**	filename{:length}, filesize{:[B/L] [1 – 8] }
– **Overwrite:**	Check to overwrite files of the same name.
– **Compress:**	Check to compress the data
– **Fail If No Source File**	Check to Fail Spray if Source File missing
– **Record Structure Present**	Check for record field names in first row

Lab Exercise 2: Fixed Spray:

1. Access your Landing Zone (AKA "Drop Zone")
Files sprayed to a THOR cluster are first placed on a Landing Zone. ECL Watch can locate and verify your Landing Zone location. To open ECL Watch, open any browser and go to *http://nnn.nnn.nnn.nnn:8010* (where *nnn.nnn.nnn.nnn* is the IP address of the ESP Server for your target HPCC cluster. Click on **Files >> Landing Zone** to access your Landing Zone.

2. Copy the file to spray to the Landing Zone
Once you know the location of your Landing Zone, you need to copy your data to that location. The **Landing Zone > Upload** section of the ECL Watch Files menu can accomplish this for you. The only issue at this time will be to confirm that you have a connection between your data source and the landing zone.

3. Access the Spray Fixed page in ECL Watch
Select (check) your file to spray ***OnlineLessonPersons*** and click on the **Spray >> Fixed** link in the top menu area .

4. Select your options to spray the *OnlineLessonPersons* Data File
Enter or verify the following settings as directed:

Group: Accept the default
Name Prefix: Specify the logical prefix name of the sprayed file. The Name must start with *ONLINE:*, followed by *your initials,* followed by *Intro::* as in this example:

ONLINE::XXX::Intro::

Target Name: Set your target name to ***Persons*.**
Record Length: The record length is ***151.***
Leave all other options blank (not checked)

5. Press **Submit** button to spray. The subsequent **DFU Workunit** page will verify that the spray completed successfully.

Other ways to spray/despray:

DFU server controls the spraying and despraying operations used to move data onto and out of THOR.

DFU services are available from:

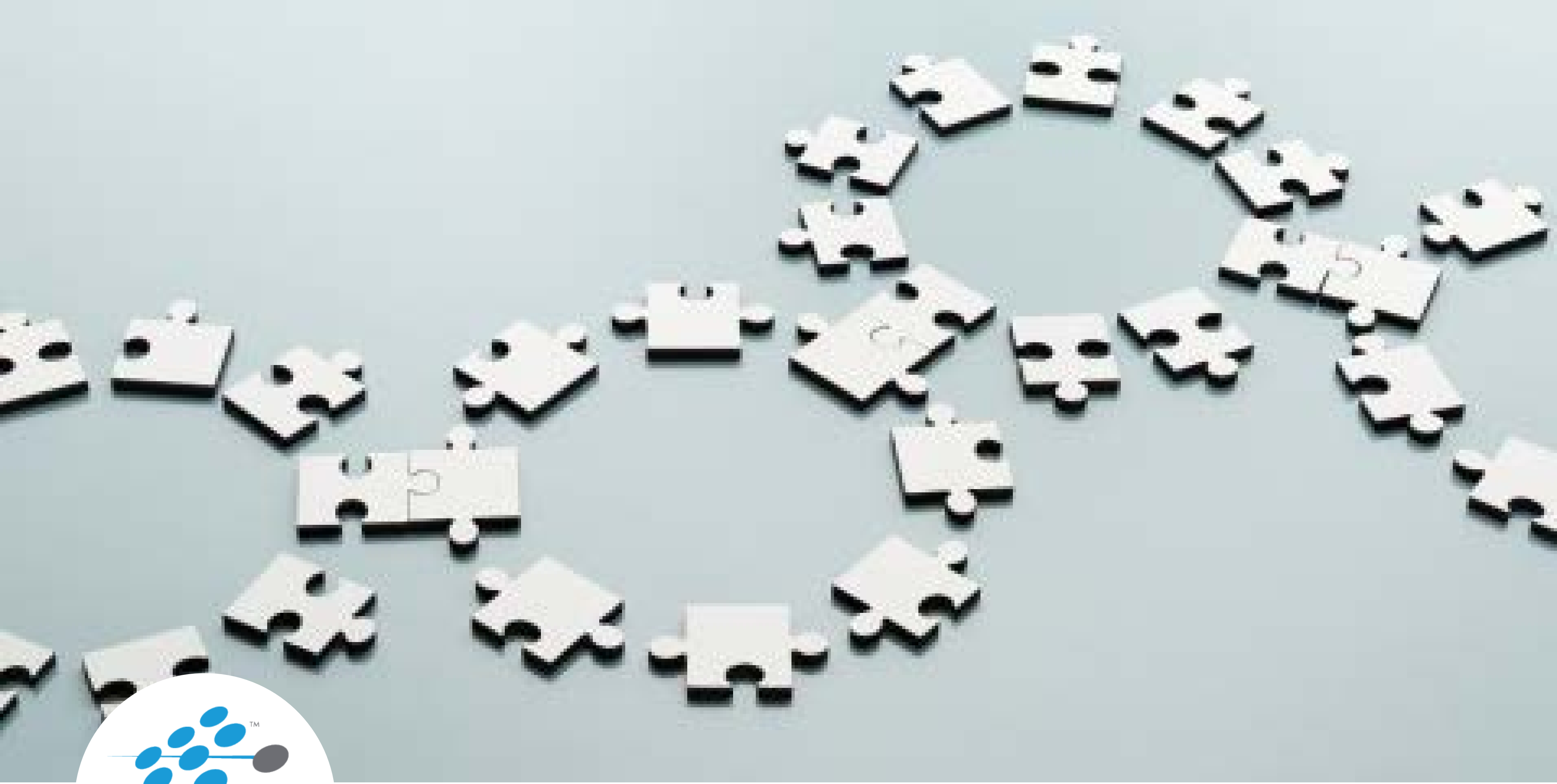
- ECL Watch.
- DFU command line interface.
- ECL IDE, using service libraries in ECL code:

```
IMPORT STD;
SrcIP    := '10.xxx.xx.200';
SrcPath  := '//10.xxx.xx.200/var/lib/HPCCSystems/mydropzone/snap-d126c9ba/';
Initials := 'BF';

//************ Spray Intro ECL/THOR Class Files *******************************

 STD.File.SprayFixed(SrcIP,SrcPath + 'persons',155, 'mythor', '~CLASS::' +
                Initials + '::Intro::Persons',,,,TRUE,TRUE);
```

Lesson Completed

Proceed to Lesson 6:

ECL Fundamentals



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 6

Fundamentals of ECL (Enterprise Control Language)

LexisNexis®

Risk Solutions

Basic ECL

A simple ECL Definition:

Name := Expression;

The **Name** must begin with a letter and may only contain letters, numbers, and underscores (_).

The ECL Definition Operator (**:=**) is read as: “is defined as”

The **Expression** must define either a single value calculation, a Boolean True/False condition, a set of values, or a set of records.

The semi-colon (**;**) is the required ECL definition terminator.

Basic ECL

Case Sensitivity / Formatting / Comments:

ECL is *not* case-sensitive. White space is ignored allowing formatting for readability.

Block comments are delimited with /* and */

Single line comments begin with //

ECL uses object.property syntax to qualify scope and disambiguate references within tables.

```
FolderName.Definition // reference a definition in another module
Dataset.Field         // reference a field in a dataset or recordset
```

Full ECL Definition Syntax

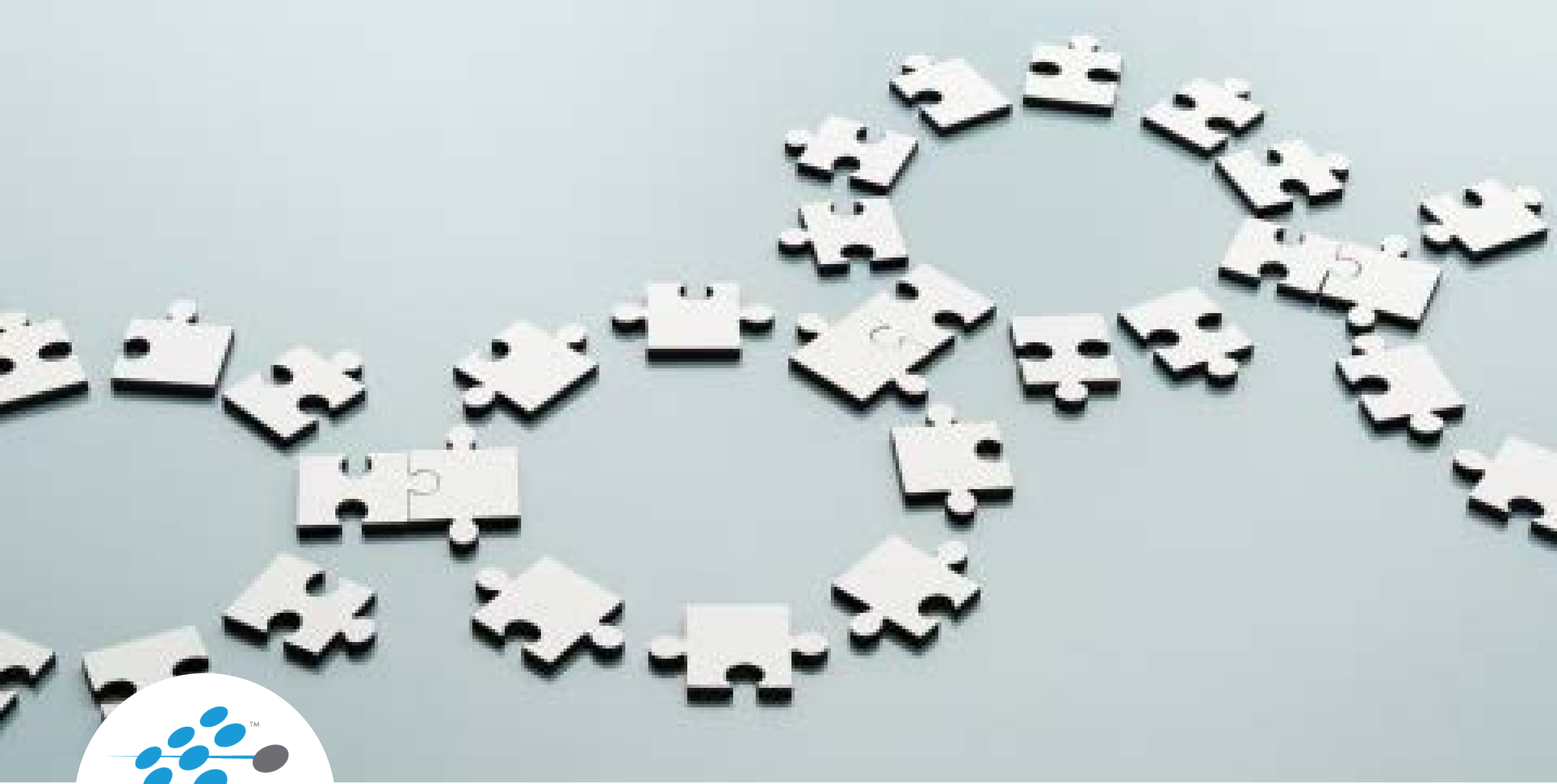
[*Scope*] [*ValueType*] **Name** [*(parameters)*] **:=**
Expression [*:WorkflowService*] **;**

```
EXPORT BOOLEAN IsLeapYear(INTEGER2 year) :=
  year % 4 = 0 AND ( year % 100 != 0 OR year % 400 = 0)
                                  : STORED('LeapYearValue');
```

Lesson Completed

Proceed to Lesson 7

Basic Definition Types



HPCC Systems:
Introduction to the Enterprise Control Language

Lesson 7

Basic Definition Types

Risk Solutions

ECL Definition Types - Boolean

Boolean –

A Boolean definition is a logical expression resulting in a TRUE/FALSE result:

```
IsGoodClass       := TRUE;
IsFloridian       := People. state = 'FL';
IsSeniorCitizen   := People.Age >= 65;
```

ECL Definition Types - Value

Value –

A Value definition is an arithmetic or string expression with a single-valued result:

```
ValueTrue         := 1;
FloridianCount    := COUNT(People(IsFloridian));
SeniorAvgAge      := AVE(People(IsSeniorCitizen), People.Age);
```

ECL Definition Types - Set

Set –

A Set definition is a set of explicitly declared constant values or expressions within square brackets (all elements must be the same type)

```
SetTrueFalseValues     := [0, 1];
SetSoutheastStates     := ['FL','GA','AL','SC'];
SetStatusCodes         := ['1','X','9','W'];
SetInts                := [1,2+3,45,def1,7*3,def2];
```

Set Ordering and Indexing

✓ ***Declared sets*** - must be indexed to access individual elements within the set

```
SetNums          := [5,4,3,2,1];

LastNumInSet     := SetNums [5];   // LastNumInSet contains the value 1
```

Indexing with Strings

Strings (character sets) - may also be indexed to access individual characters within the string

```
MyString    := 'ABCDE';
MiddleChar := MyString[3];      //MiddleChar contains “C”
```

Substrings - may be extracted by using 2 periods to separate the beginning and ending element numbers to specify the substring to extract

```
MySubString1 := MyString[2..4];  // 'BCD'
MySubString2 := MyString[..4];    // 'ABCD'
MySubString3 := MyString[2..];    // 'BCDE'
```

ECL Definition Types - Recordset

<u>Recordset</u> –

A Recordset definition is a filtered dataset or recordset

```
FloridaPeople          := People(IsFloridian);
SeniorFloridaPeople    := FloridaPeople(IsSeniorCitizen);
```

Recordset Ordering and Indexing

Recordsets - may also be indexed to access individual records (or ranges) within the recordset or dataset

```
MySortedRecs := SORT(People(LastName = 'TAYLOR'), FirstName);

FirstTaylor   := MySortedRecs[1];      //First record in the sorted set
FirstTaylors := MySortedRecs[1..3];  //First three records
```

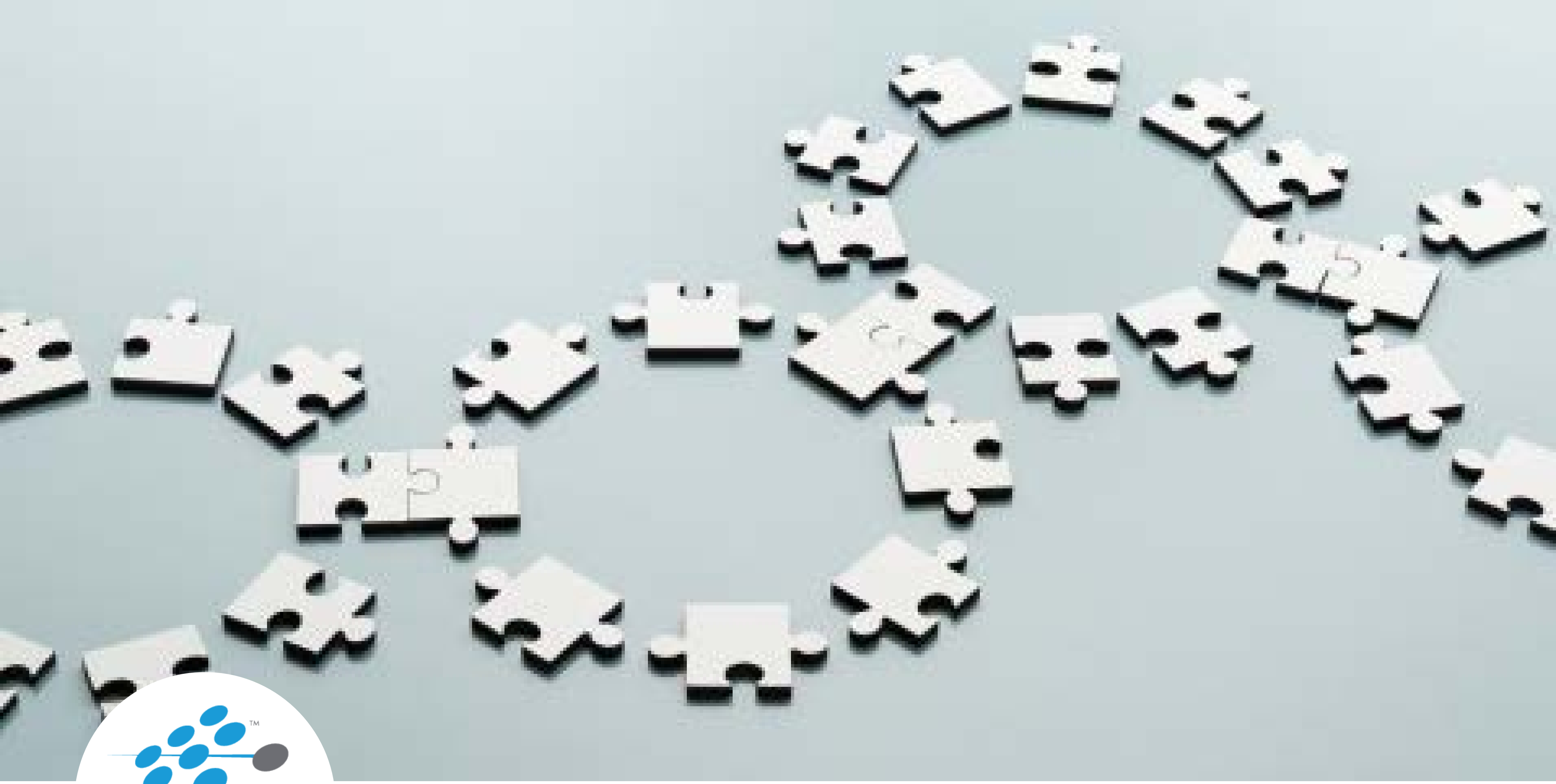
Fields in Records - may be accessed using indexing to access the individual record then qualifying the field to return

```
FirstTaylorFirstName := MySortedRecs[1].FirstName;
            //First name in first record in the sorted set
```

Lesson Completed

Proceed to Lesson 8:

Datasets-Recordsets-Filters



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 8

Datasets, Record Sets, and Filters

LexisNexis®

Risk Solutions

Datasets, Record Sets, and Filters

Physical **Datasets** can be *memory* or *disk* based:

```
//Physical DATASET example
EXPORT Persons := DATASET('~ONLINE::XXX::Intro::Persons',
                          Layout_Persons,THOR);

//Inline (In Memory) DATASET example:
SomeFile := DATASET([{'C','G'},{'C','C'},{'A','X'},{'B','G'},{'A','B'}],
                    {STRING1 Value1,STRING1 Value2});
```

Datasets, Record Sets, and Filters

Record Sets are *subsets* of physical datasets or other record sets based on *filtering* conditions.

The term "Record Set" indicates any set of records derived from a Dataset (or another Record Set), usually based on some filter condition to limit the result set to a subset of records. Record sets are also created as the return result from one of the built-in functions that return result sets.

When a record set is used as any parameter in an ECL Function, the usage of "recordset" (removing the space) is permitted.

Datasets, Record Sets, and Filters

Filters are specified in parentheses following the dataset or record set name in any expression:

```
Lifestyle_File := Polk.File_Polk_TLS((INTEGER)DOB > 191604 AND
                                     (INTEGER)DOB <= 196204 AND
                                    DOB <> '');
```

Record Set Filters

Any Boolean expression within parentheses following a Dataset or Record Set name is a **filter expression**, used to define the specific subset of records to use.

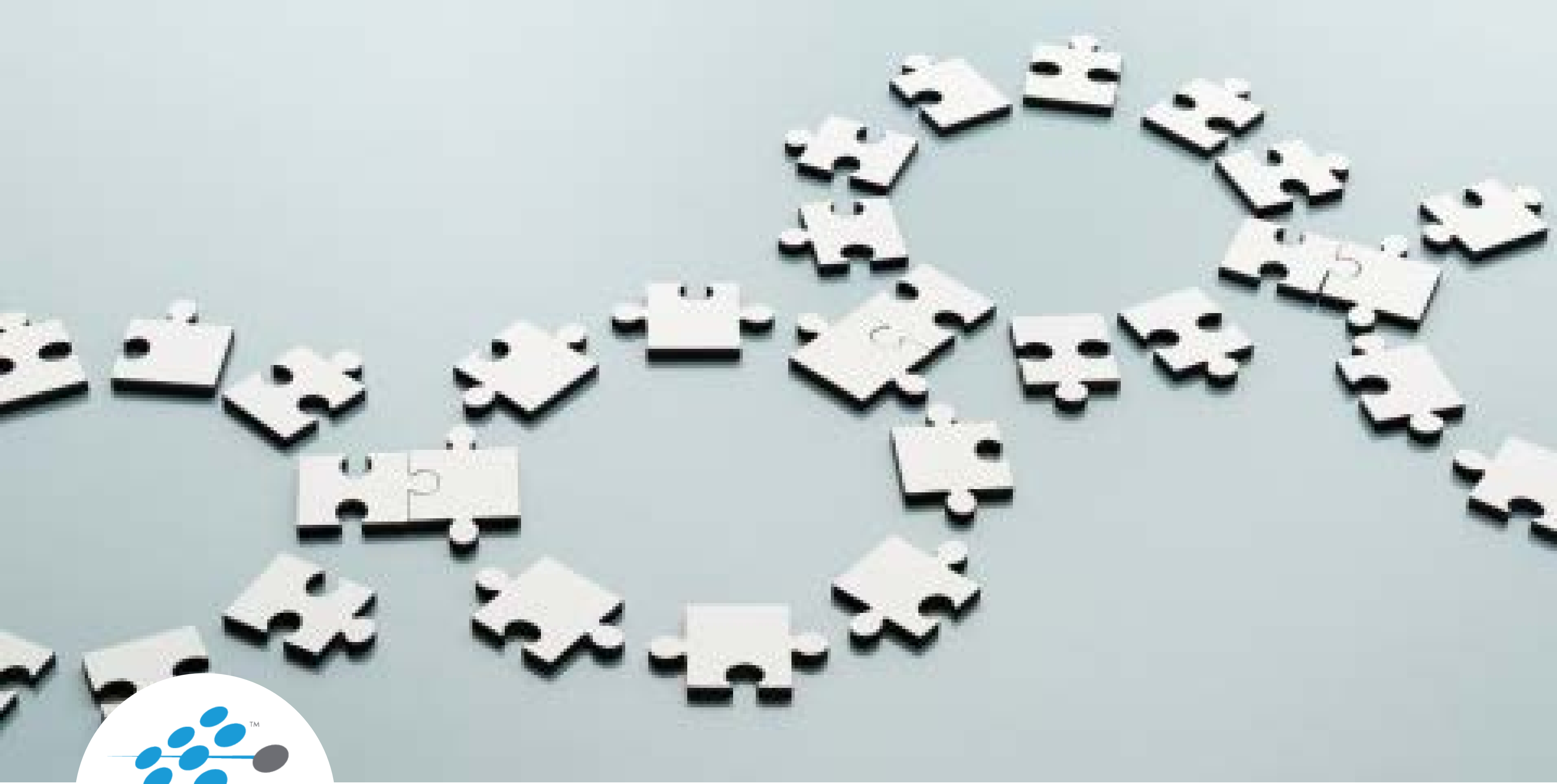
Multiple filter conditions may be specified by separating each filter expression with a comma (,) or explicitly using the AND operator

```
T_Names := People(Lastname >= 'T', Lastname < 'U');

RateGE7Trds := Trades(trd_rate >= 7);

ValidTrades := Trades(NOT IsMortgage AND NOT IsClosed);
```

Lesson Completed

Proceed to Lesson 9:

ECL Functions and Scoping



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 9

ECL Functions and Scoping

LexisNexis®

Risk Solutions

Functions and Parameter Passing

Any ECL Definition can be defined to accept passed parameters (arguments), making it an **ECL Function**, which does not change the definition's basic type, just makes it more flexible.

Defined parameters always appear in parenthesis following the ECL definition name.

Multiple parameter definitions are separated by commas.

Functions and Parameter Passing

DefinitionName([*ValueType*] *ParameterName* [=*DefaultValue*]) := *expression*;

The required *ParameterName* names the parameter passed for use in the *expression*

The optional *ValueType* defaults to INTEGER if omitted

The optional *DefaultValue* makes the parameter omittable

Functions and Parameter Passing – Example:

```
Num25 := 25;
AddFive(INTEGER x=10) := x + 5;

NumResult1 := AddFive(Num25);   // Result is 30
NumResult2 := AddFive();        // Result is 15

//Passing a DATASET as a parameter
MyRec      := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],MyRec);

FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
            //passed dataset referenced as "ds" in expression
OUTPUT(FilteredDS(SomeFile));
```

ECL Definition Visibility (Scoping)

Global (sort of) – An ECL definition with the EXPORT keyword is available “globally” throughout its own module and in any other module that IMPORTs its module (must be fully qualified)

```
EXPORT PeopleCount := COUNT(People); //Available “globally”
```

Module – An ECL definition with the SHARED keyword is only available throughout its own module

```
SHARED StateCount := 50;   //Available only within its own module
```

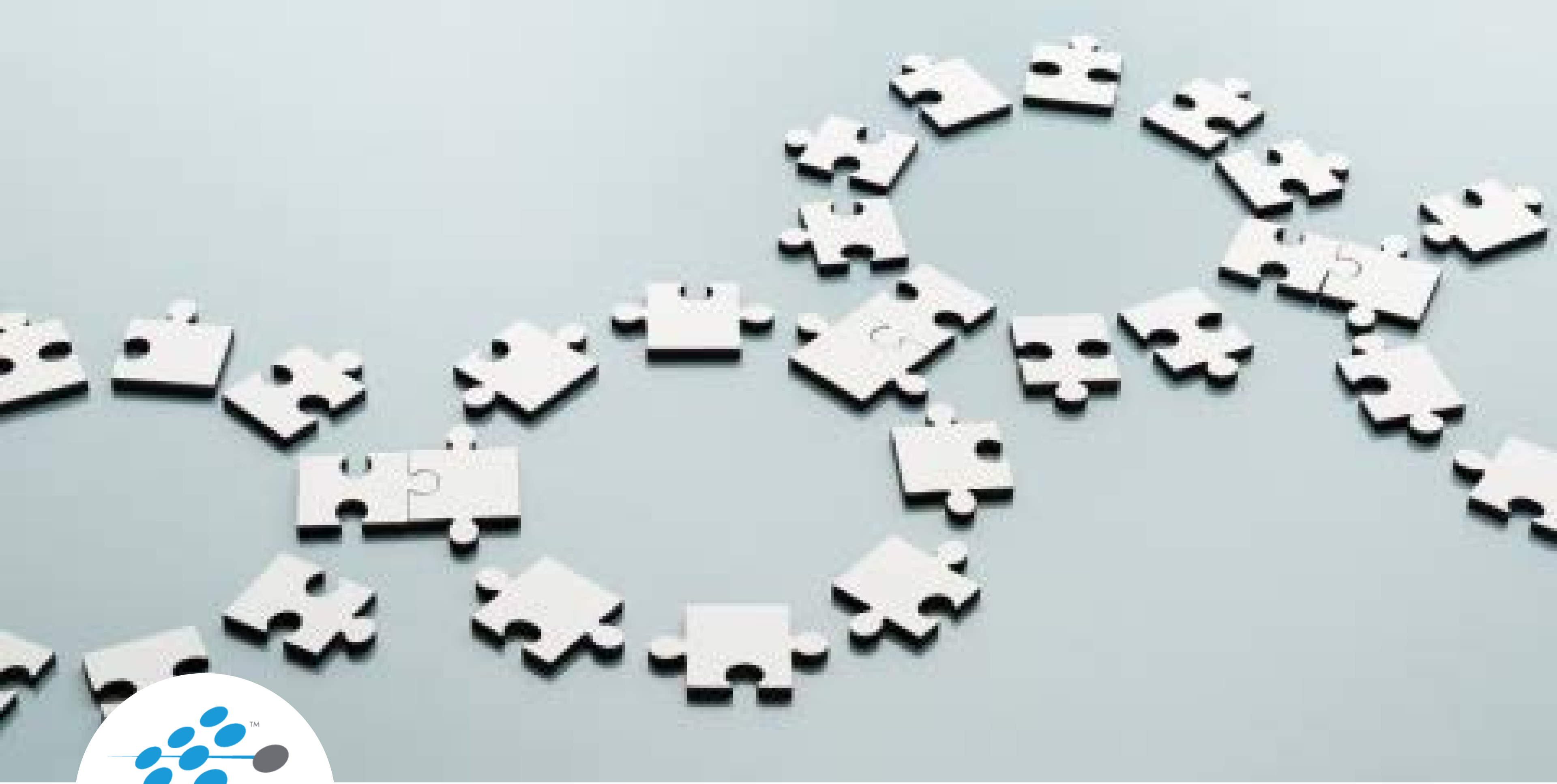
Local – An ECL definition without EXPORT or SHARED is available only within other EXPORT or SHARED definitions

```
Num5 := 5;              // Local Scope – Available only to NumTotal
EXPORT NumTotal := Num5 + 10 + StateCount;
```

Lesson Completed

Proceed to Lesson 10:

Qualifications and Actions



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 10

Qualification and Actions

Risk Solutions

Qualification

ECL Definition Qualification –

EXPORTed definitions in other folders and IMPORTed are available for use when qualified by the folder name

```
IMPORT Companies; // Exported definitions in Companies available
FloridaCompanies := Companies.File_Company(state='FL');

IMPORT $; // Exported definitions in current folder available
FloridaCompanies := $.File_Company(state='FL');

IMPORT * FROM Companies;
// All exported definitions in Companies available
FloridaCompanies := File_Company(state='FL');
//This usage is not recommended;
```

Qualification (continued)

Data Field Qualification – Fields within datasets are qualified by prepending the dataset or record set name

```
Young := AgeOf(People.dob) < 30; //field in the Person dataset
```

Implicit Data Field Qualification –Dataset or Recordset fields come into scope and can be referenced without qualification in filters

```
YoungPeople := People(dob[1..6] > '197211');
        //dob doesn’t need to be qualified here because
        // it’s used to filter the dataset to which it belongs
```

Actions and Definitions

1. **Actions instigate Workunits, Definitions don't**

```
OUTPUT(People); // will instigate a Workunit (WU)
People;         // implicit action, will instigate a Workunit
```

2. **Actions can be ECL definitions**

```
A := OUTPUT(People);   // will NOT instigate a Workunit
```

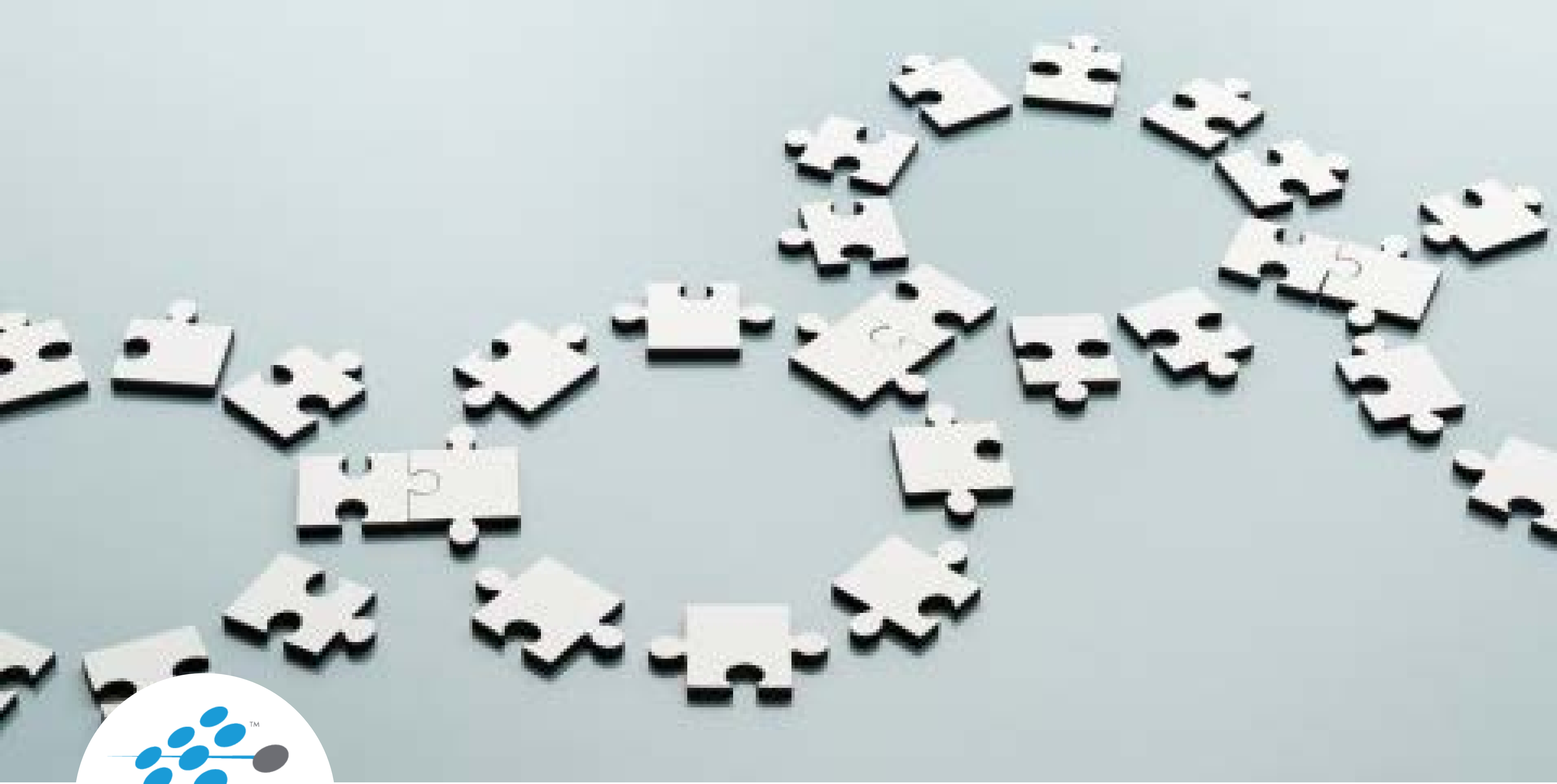
3. **Functions that return scalar values can act as Actions**

```
A := COUNT(People); // definition will not instigate a WU
COUNT(People);      // WILL instigate a Workunit
```

Lesson Completed

Proceed to Lesson 11:

Operators and Expressions



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 11

Operators and Expressions

Risk Solutions

Arithmetic Operators:

Division	/
Integer Division	DIV
Modulus Division	%
Multiplication	*
Addition	+
Subtraction	-

Note: Any division by zero (0) results in zero (0).

Bitwise Operators:

Bitwise AND	&
Bitwise OR	|
Bitwise Exclusive OR	^
Bitwise NOT	BNOT
Bitshift Right	>>
Bitshift Left	<<

Comparison Operators:

Equivalence	=	
Not Equal	<>	
Not Equal	!=	
Less Than	<	
Greater Than	>	
Less Than or Equal	<=	
Greater Than or Equal	>=	
Equivalence Comparison	<=>	returns -1, 0, or 1

More Operators:

- ✓ **Record Set Operators**

  Append +

  CompCombined1 := Comp1 **+** Comp2 **+** Comp3;

  Append & (maintains record order on each node)

  CompCombined2 := Comp1 **&** Comp2 **&** Comp3;

- ✓ **Set Operators**

  Append +

  SetCombined := [1,2,3] **+** [4,5,6] **+** [7,8,9];

- ✓ **Logical Operators**

NOT	Boolean NOT operation
~(tilde)	Boolean NOT operation
AND	Boolean AND operation
OR	Boolean OR operation

Operators and Expressions:

String Operators

Concatenation +

```
FullName := TRIM(FirstName) + ' ' + LastName;
```

Expressions:

Expression Evaluation

Expressions are evaluated left-to-right and from the inside out (in nested functions). Parentheses may be used to alter the default evaluation order of precedence for all operators.

Operator evaluation order of precedence is as listed in the *ECL Language Reference*

Expressions: IN Operator

- ✓ **IN Operator**

  value **IN** *value_set*

  - ✓ *value* – The value definition or constant to search for in the *value_set*.
  - ✓ *value_set* – The set to search.

  The **IN** operator is a shorthand for a collection of OR conditions. It searches the *value_set* to find a match for the *value* and returns a Boolean TRUE or FALSE.

```
SetSoutheastStates := ['FL','GA','AL','SC'];
BOOLEAN IsSoutheastState(STRING2 state) :=  state IN SetSoutheastStates;
```

Expressions: BETWEEN Operator

- ✓ **BETWEEN Operator**

  seekval **BETWEEN** *loval* **AND** *hival*

  - ✓ *seekval* – The value to find.
  - ✓ *loval* – The low value in the inclusive range.
  - ✓ *hival* – The high value in the inclusive range.

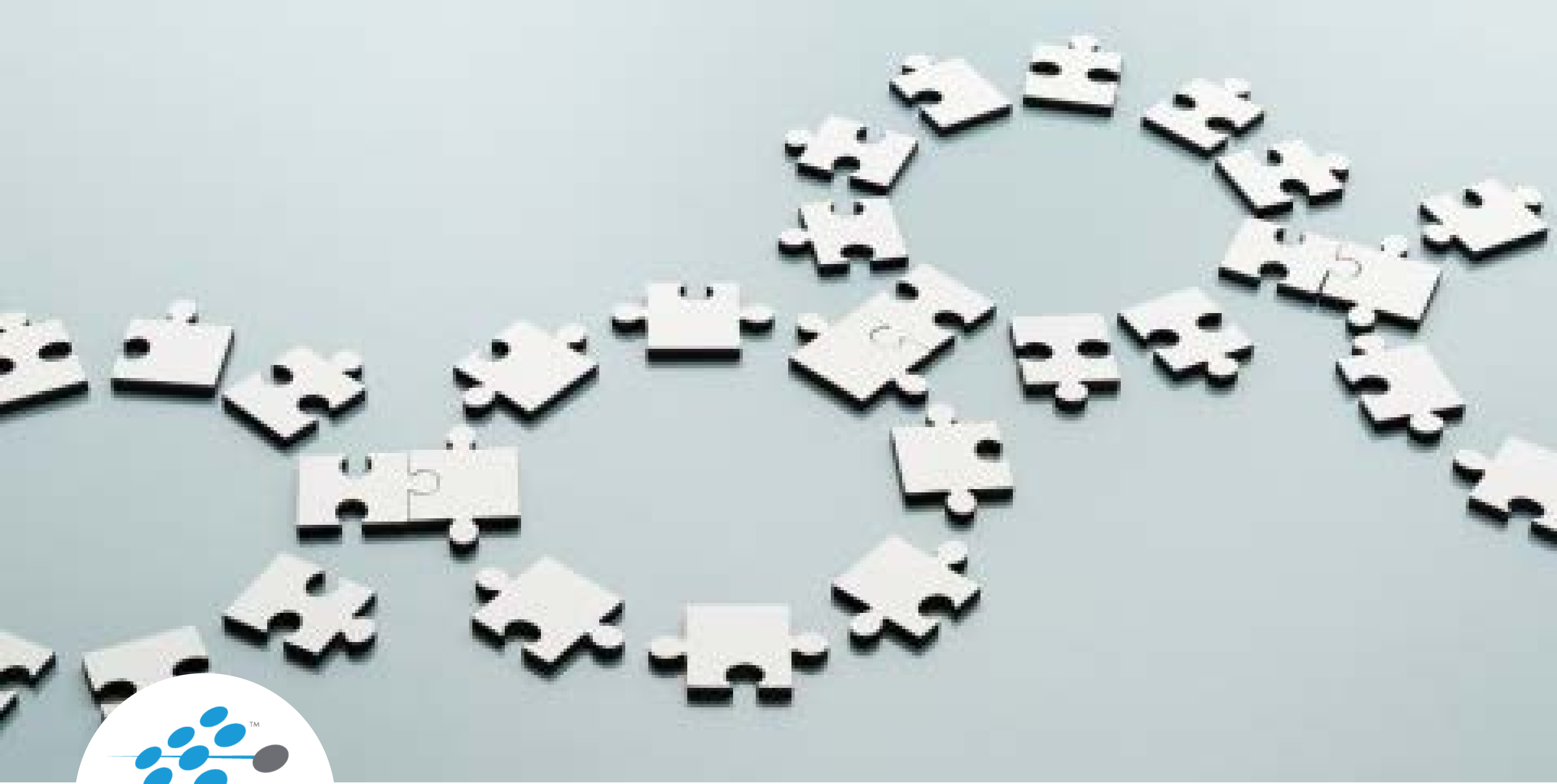
  The **BETWEEN** operator is shorthand for an inclusive range check using standard comparison operators (*SeekVal* >= *LoVal* AND *SeekVal* <= *HiVal*). It may be combined with NOT to reverse the logic.

```
IsOne2Ten(val) := val BETWEEN 1 AND 10;
  // is the passed val in the inclusive range of 1 to 10?
```

Lesson Completed

Proceed to Lesson 12:

Value Types



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 12

Value Types

LexisNexis®

Risk Solutions

Value Types - BOOLEAN

A Boolean true/false expression

TRUE and FALSE are reserved ECL keywords and Boolean constants that may be compared to a Boolean type

```
BOOLEAN NBEQ(STRING L, STRING R) :=
                        L <> '' AND R <> '' AND L=R;

BOOLEAN IsLeapYear(INTEGER2 year) :=
              year % 4 = 0 AND
              ( year % 100 != 0 OR year % 400 = 0);
```

Value Types – SIGNED/UNSIGNED INTEGERS

[*IntType*] [**UNSIGNED**] **INTEGER**[*n*]

[*IntType*] **UNSIGNED***n*

- ✓ An *n*-byte integer value
- ✓ Valid *n* values are 1 to 8
- ✓ *IntType* can be BIG_ENDIAN or LITTLE_ENDIAN (LITTLE_ENDIAN is the default)
- ✓ If the UNSIGNED keyword is specified the value is always positive, otherwise it may be negative
- ✓ INTEGER8 is the default if no size *n* is specified

```
UNSIGNED1 ctr    := 0;           // 0 – 255
INTEGER1 ictr    := -100;        // -128 to 127
INTEGER BigNum := 9223372036854775807;
                                  // 9,223,372,036,854,775,807
```

Value Types - REAL

REAL[*n*]

- ✓ An *n*-byte standard IEEE floating point value. Valid *n* values are 4 (up to 7 significant digits) and 8 (up to 15 significant digits)
- ✓ REAL8 is the default if no size *n* is specified

```
REAL4 PI := 3.14159;

REAL8 TotalBudget := 153347820.75;
```

Value Types – DECIMAL/UDECIMAL

[UNSIGNED] DECIMAL*n*[_*y*]

UDECIMAL*n*[_*y*]

- ✓ A packed decimal value of *n* total digits (to a maximum of 32) -- if the _*y* value is present, the *y* defines the number of decimal places in the value
- ✓ If the UNSIGNED keyword is omitted, the rightmost nibble holds the sign
- ✓ Using exclusively DECIMAL values in computations invokes BCD math libraries (base-10 math), allowing up to 32 digits of precision (which may be on either side of the decimal point)

```
DECIMAL7_2 Salary := 75000.00;
DECIMAL6_5 NegativeLeaseFactor := -0.23544;
```

Value Types - STRING

[*StringType*] **STRING**[*n*]

- ✓ A character string of *n* bytes, space padded (not null terminated)
- ✓ If *n* is omitted, the string is variable length to the size needed to contain the result of a cast or passed parameter
- ✓ Optional *StringType* may be ASCII or EBCDIC (the default is ASCII)

```
STRING1 Gender := 'M';
EBCDIC STRING20 CityName := 'BOCA RATON';
```

Value Types - QSTRING

QSTRING[*n*]

- ✓ A variation of STRING using only 6-bits of storage per character, reducing storage requirements for large strings
- ✓ Character set is limited to the capital letters A-Z, 0-9, spaces, and the following special characters:

  ! “ # $ % & ‘ ( ) * + , - . / ; < = > ? @ [\ ] ^ _
- ✓ If *n* is omitted, the QSTRING is variable length to the size needed to contain the result of a cast or passed parameter

```
QSTRING15 CompanyName := ‘LEXISNEXIS, INC';
  // uses 12 bytes of storage instead of 15
```

Value Types - UNICODE

UNICODE[*n*]

- ✓ A UTF-16 encoded character string
- ✓ Space padded
- ✓ If *n* is omitted, the UNICODE is variable length to the size needed to contain the result of a cast or passed parameter

UNICODE20 CompanyName := U' LEXISNEXIS';

Value Types - DATA

DATA[*n*]

✓ A packed hexadecimal character string of *n* bytes, zero padded (not space padded)

DATA9 CID := x'1a2d33548704662988';

Value Types - VARSTRING

VARSTRING[*n*]

- ✓ A null-terminated character string containing *n* bytes of data
- ✓ If *n* is omitted, the VARSTRING is variable length to the size needed to contain the result of a cast or passed parameter

```
VARSTRING10 city := 'BOCA RATON';
      // uses 11 bytes of storage
```

Value Types - VARUNICODE

VARUNICODE[*n*]

- ✓ A UTF-16 encoded character string
- ✓ Null terminated
- ✓ If *n* is omitted, the VARUNICODE is variable length to the size needed to contain the result of a cast or passed parameter

VARUNICODE20 CompanyName := U' LEXISNEXIS';

TYPE Structure

ALIEN Data Type

```
typename := TYPE
              function;
            END;
```

- ✓ *typename* – The name of the TYPE structure.
- ✓ *function* – Special function ECL definitions. These include LOAD, STORE, PHYSICALLENGTH, GETISVALID, MAXLENGTH that define how the data type is manipulated (see the *ECL Language Reference*).

The **TYPE** structure defines a series of functions that are implicitly invoked when the *typename* is subsequently used in a RECORD structure as a value type. Parameters may be passed to the TYPE structure definition which may then be used in any of the function definitions. TYPE allows you to define most any "alien" data type not already supported in ECL by defining how to load, store, and validate the "alien" type.

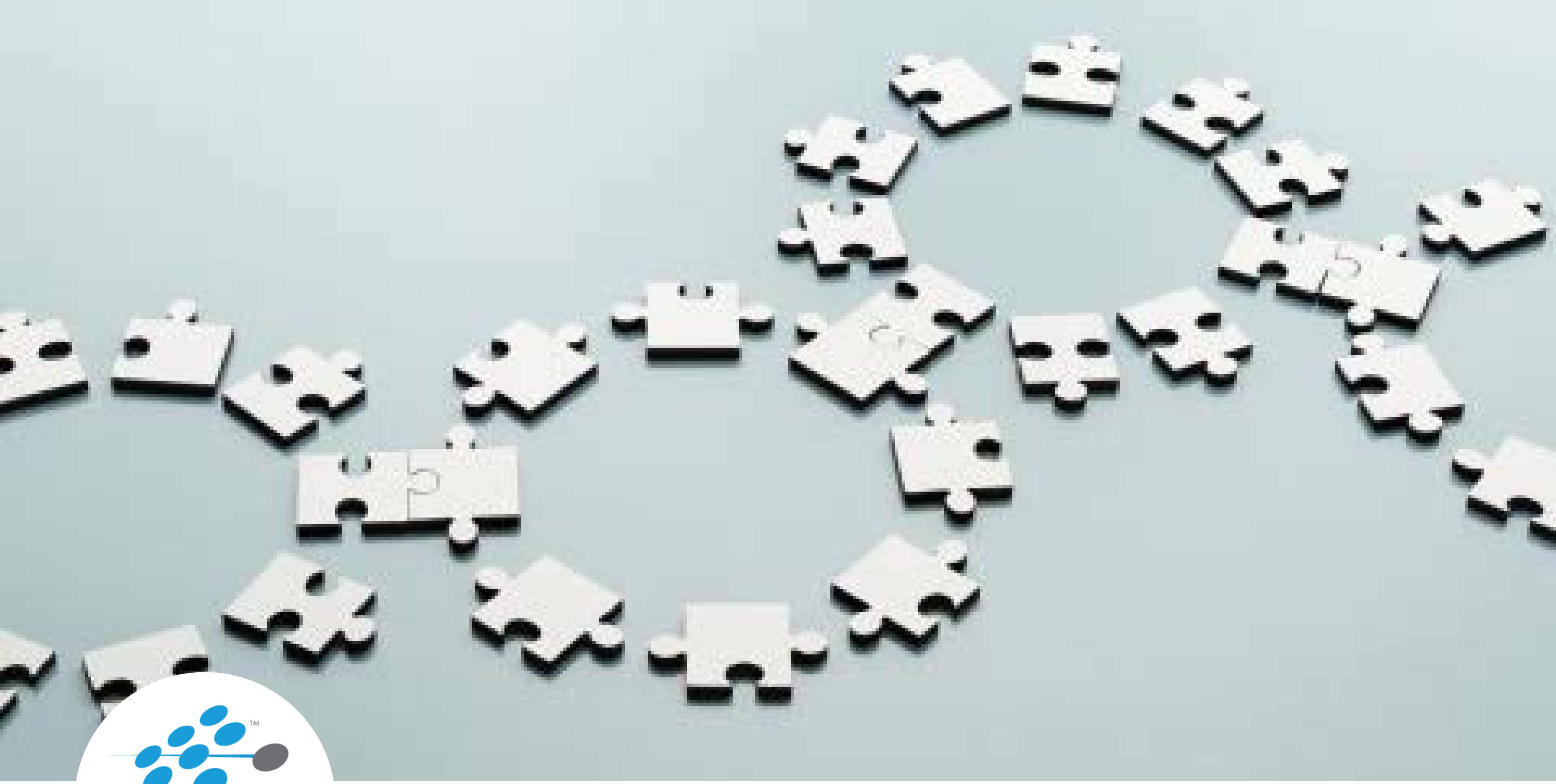
TYPE Example:

```
EXPORT dstring(STRING del) := TYPE
        EXPORT INTEGER PHYSICALLENGTH(string s) :=
              StringLib.StringFind2(s,del)+LENGTH(del)-1;
  EXPORT STRING LOAD(STRING s)  :=
              s[1..StringLib.StringFind2(s,del)-1];
        EXPORT STRING STORE(STRING s) := s + del;
END;
EXPORT Layout_Tranfile := RECORD
   dstring(':') Orig_Id;
   dstring(':') Orig_Type;
   dstring(':') Dest_Id;
   dstring(':') Dest_Type;
   dstring(':') date_stamp;
   dstring(':') time_stamp;
   dstring('\n') dest_field;
END;
```

Lesson Completed

Proceed to Lesson 13:

ECL Constants, More Value Types and Casting Rules



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 13

ECL Constants, Value Types and Casting Rules

Risk Solutions

ECL Constants

HPCC SYSTEMS®

- ✓ *String* – text enclosed by single quotation marks ' '
  (\' to include a single quote, \\ to include a backslash)
  ```
  STRING20 MyName := 'Tony Middleton';
  ```
- ✓ *Hexadecimal String* – Leading “X” with hex value in single quotes
  ```
  DATA2 CRLF := x'0D0A';
  ```
- ✓ *Data String* – Leading “D” with text in single quotes
  ```
  MyData := D'abcd';
  ```
- ✓ *Unicode String* – Leading “U” with text in single quotes
  ```
  MyUniString := U'abcd';
  ```
- ✓ *VARSTRING* – Leading “V” with text in single quotes
  ```
  MyVarString := V'abcd';
  ```
- ✓ *QSTRING* – Leading “Q” with text in single quotes
  ```
  MyQString := Q'ABCD';
  ```

Numeric Constants

✓ *Numeric* – numeric constants containing a decimal portion are treated as REAL values and those without are treated as INTEGER. Integer constants may be decimal, hexadecimal or binary values.

```
MyInt := 10;          // value is INTEGER 10
MyInt := 0x0A;        // value is INTEGER 10
MyInt := 0Ax;         // value is INTEGER 10
MyInt := 0b1010;      // value is INTEGER 10
MyInt := 1010b;       // value is INTEGER 10

MyReal := 10.0;       // value is REAL 10.0
```

Value Types - SET

SET [OF *type* **]**

✓ *type* – The value type of the set (INTEGER, REAL, BOOLEAN, STRING, UNICODE, DATA, or DATASET(recstruct)). If omitted, defaults to INTEGER.

SET and **SET OF** define the name or passed parameter as a set of values. The keyword ALL may be used as the passed parameter default value to indicate all possible values for the set, or empty square brackets to indicate no possible value.

```
SET OF INTEGER1 SetCloseCodes := [65,66,90,114,115,123];
HasCode(INTEGER code, SET s) := code IN s;
ClosedAccounts := Accounts(HasCode(acct_status,SetCloseCodes));
```

TYPEOF

TYPEOF(*expression***)**

- ✓ *expression* – An expression defining the value type. This may be the name of a data field, passed parameter, function, or definition providing the record type including RECORD structures.

  The **TYPEOF** declaration allows you to define a name or parameter whose value type is the same as the passed *expression*. Typically used as the result type of a TRANSFORM function.

```
INTEGER4 Num := 9765;

TYPEOF(Num) Num1 := 100567;
```

RECORDOF

RECORDOF(*recordset*)

- ✓ *recordset* – The set of data records whose RECORD structure to use. This may be a DATASET or any derived recordset.

The **RECORDOF** declaration specifies inheriting just the record layout (without default values) of the specified *recordset*.

```
t := TABLE(People,{LastName,FirstName});

r := RECORD
  RECORDOF(t);
  UNSIGNED1 NewByte;
END;
```

ENUM

ENUM([*type* **,]** *name* **[=***value***] [** *, name* **[=***value***] ...])**

- ✓ *type* – The numeric type of the *values* (defaults to UNSIGNED4).
- ✓ *name* – The label for the enumerated *value*.
- ✓ *value* – The numeric value to associate with the *name*. If omitted, each *value* is the previous *value* plus one.

The **ENUM** type specifies constant values to make code more readable.

```
Gender := ENUM(UNSIGNED1,Male,Female,Neutral,Unknown);
```

Type Casting

Explicit Casting – place the value type in parentheses in the expression immediately preceding the element to cast.

```
STRING10 Value    := '34658';
Cnt               := (INTEGER4) Value;
StrCnt            := (STRING5) Cnt;
```

Implicit Casting – During expression evaluation, different value types may be implicitly cast in order to evaluate the expression, promoting one value type to another. STRING to INTEGER, INTEGER to REAL.

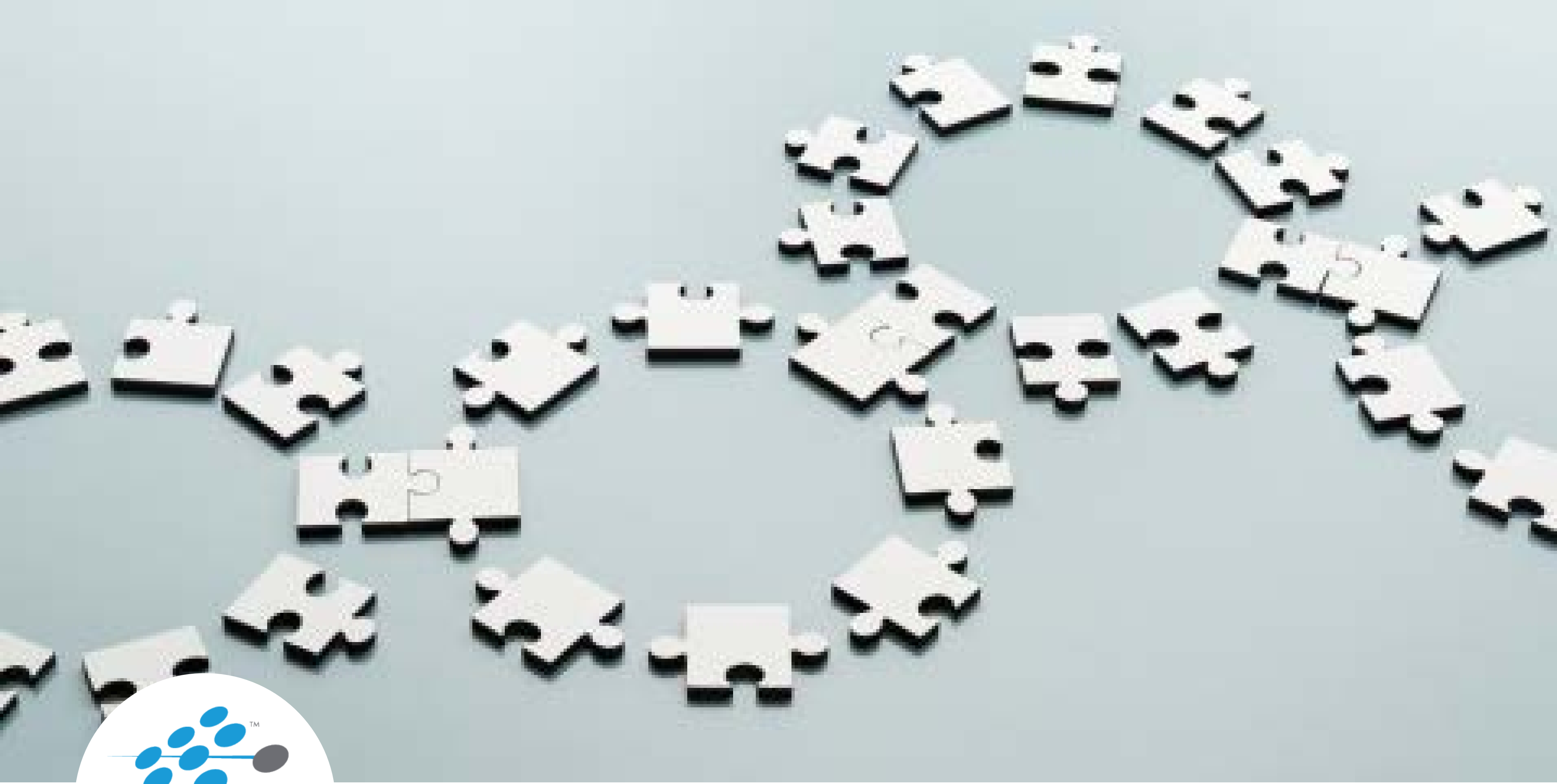
Type Transfer – treats the bit pattern as different type

Casting Rules – see the *ECL Language Reference*

Lesson Completed

Proceed to Lesson 14:

RECORD and DATASET



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 14

RECORD and DATASET

LexisNexis®

Risk Solutions

Rules for working with data on the HPCC

Before you can work on any data entity on the HPCC Cluster, three steps must be followed:

1. Spray
2. Define (RECORD)
3. Locate or Identify (DATASET)

RECORD Structure

DefinitionName := RECORD

fields;

END;

- ✓ *DefinitionName* – The name of the RECORD structure.
- ✓ *fields* – The data type and name of each data field.

A **RECORD** structure defines the field layout of a DATASET, recordset, INDEX, or TABLE. The keywords RECORD and END may be replaced with the open and close curly braces ({}) and the semi-colon field delimiters may be replaced with commas.

RECORD Structure Example:

```
EXPORT Layout_Company := RECORD
  UNSIGNED      sic_code;
  STRING1       source;
  STRING120     company_name;
  STRING10      prim_range;
  STRING2       predir;
  STRING28      prim_name;
  STRING4       addr_suffix;
  STRING2       postdir;
  STRING5       unit_desig;
  STRING8       sec_range;
  STRING25      city;
  STRING2       state;
  STRING5       zip;
  STRING4       zip4;
  STRING10      phone;
END;
```

DATASET

- ✓ ***name*** **:= DATASET(** *file, recorddef,* **THOR** *thoroptions***);**
  name **:= DATASET(** *file, recorddef,* **CSV [(** *csvoptions* **)]);**
  name **:= DATASET(** *file, recorddef,* **XML(** *path* **));**
  name **:= DATASET(** *file, recorddef,* **PIPE(** *command* **));**
  - ✓ *name* – The definition name by which the file is subsequently referenced.
  - ✓ *file* – A string constant containing the logical filename.
  - ✓ *recorddef* – The RECORD structure of the dataset.
  - ✓ *thoroptions* – options specific to THOR/FLAT datasets.
  - ✓ *command* – third-party program that creates the dataset.

  DATASET introduces a new data file into the system with the specified *recorddef* layout.

RECORD and DATASET example

```
Layout_Company := RECORD
  UNSIGNED      sic_code;
  STRING120     company_name;
  STRING10      prim_range;
  STRING2       predir;
  STRING28      prim_name;
  STRING4       addr_suffix;
  STRING2       postdir;
  STRING5       unit_desig;
  STRING8       sec_range;
  STRING25      city;
  STRING2       state;
  STRING5       zip;
  STRING4       zip4;
END;

EXPORT File_Company_List := DATASET('~CLASS::Company_List', Layout_Company, THOR);
```

INLINE DATASET

name := **DATASET(***recordset*, *recorddef***)**

- ✓ *name* – The definition name by which the file is subsequently referenced.
- ✓ *recordset* – A set of data records contained within square brackets (indicating a set definition). Within the square brackets, each record is delimited by curly braces ({}) and separated by commas. The fields within each record are comma delimited.
- ✓ *recorddef* – The RECORD structure of the dataset.

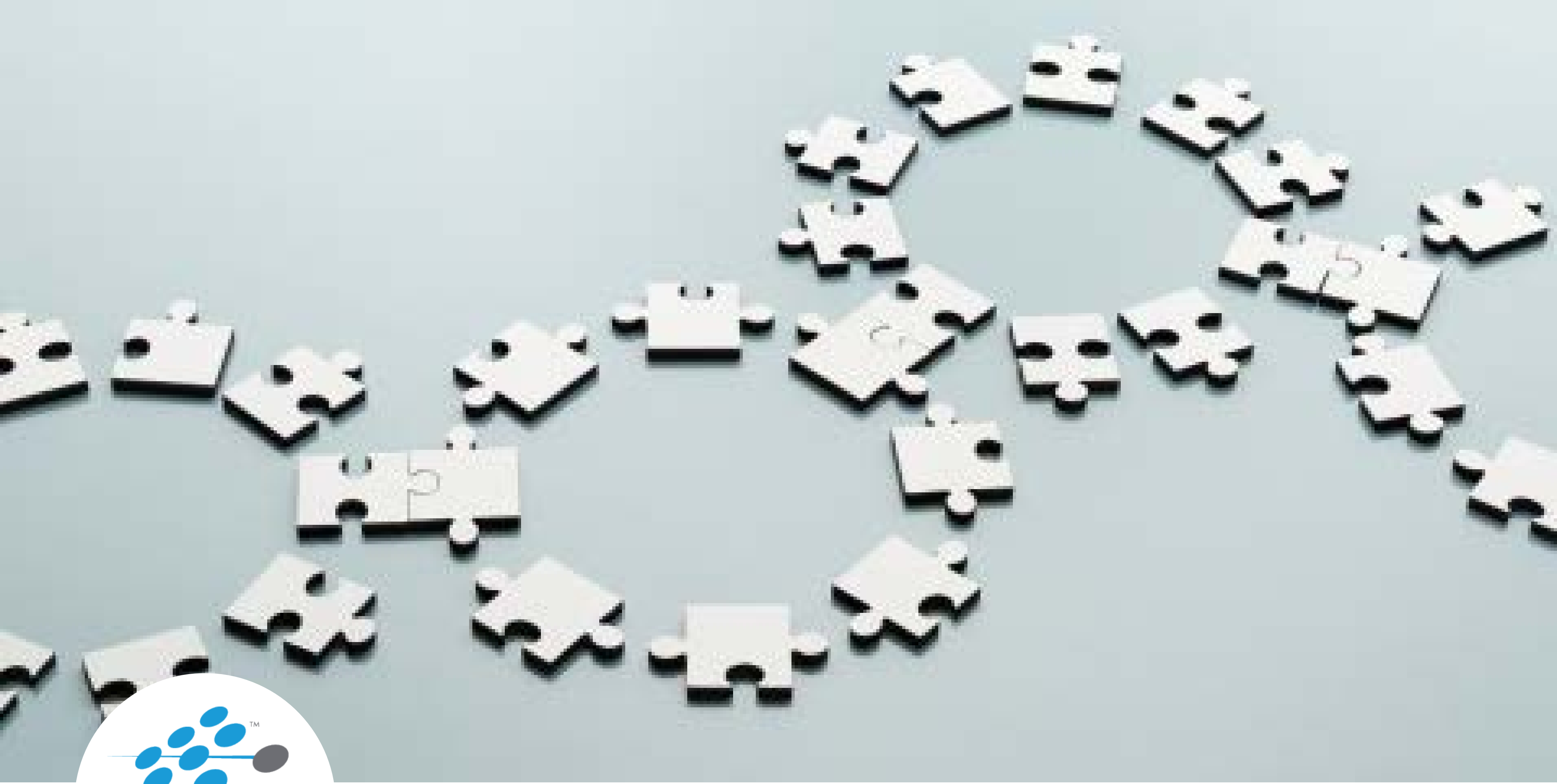
DATASET introduces a new table into the system with the specified *recorddef* layout. This form allows you to treat an inline set of data as a data file.

```
NamesRec := RECORD
    STRING20 first_name;
    STRING20 last_name;
END;
Names := DATASET([{'John','Jones'}, {'Jane','Smith'}], NamesRec);
```

Lesson Completed

Proceed to Lesson 15:

ECL Scoping Review



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 15

ECL Scoping Review

LexisNexis®

Risk Solutions

Scope and Logical Filenames

Filenames always start with a scope, followed by the directory structure, and terminating with the name of the file itself. The DFU looks for a file whose name starts with the default scope, followed by the supplied filename:

```
'DIR1::DIR2::FileName'           //given this, DFU looks for:
'SCOPE::DIR1::DIR2::FileName'    //this file
```

Leading tilde (~) indicates an overridden default scope

```
'~SCOPE::DIR1::DIR2::FileName'   //given this, DFU looks for:
'SCOPE::DIR1::DIR2::FileName'    //this file
```

ECL Repository Folders

- ✓In the ECL Repository, ECL definition files are stored in Folders
- ✓Folder names must be valid definition names
- ✓ECL definitions that reference other definitions in a different folder must include an IMPORT statement with that folder name
- ✓ECL definitions within a folder that can be referenced from other folders must begin with the EXPORT keyword
- ✓References to definitions in another folder use *foldername.definitionname* syntax

Reserved Keywords - IMPORT

IMPORT *folderlist*

- ✓ *folderlist* – A comma-delimited list of folder names. This may contain the keyword AS followed by an alias to rename a specific folder in the list.

The **IMPORT** keyword defines a list of folders whose exported definition files are available for use in the current definition text control.

```
IMPORT $; //Alias current folder name with $
IMPORT * FROM TrainingYourName; //override qualification
IMPORT Companies, STD;
IMPORT  YellowPages AS YP;
```

Reserved Keywords - EXPORT

EXPORT *definition*

- ✓ *definition* – A definition file in a repository folder.

The **EXPORT** keyword explicitly allows other folders to import the specified *definition* file.

```
EXPORT File_Company := DATASET('Company',
                               Layout_Common,FLAT);
```

Reserved Keywords - SHARED

SHARED *definition*

- ✓ *definition* – An ECL definition within a file.

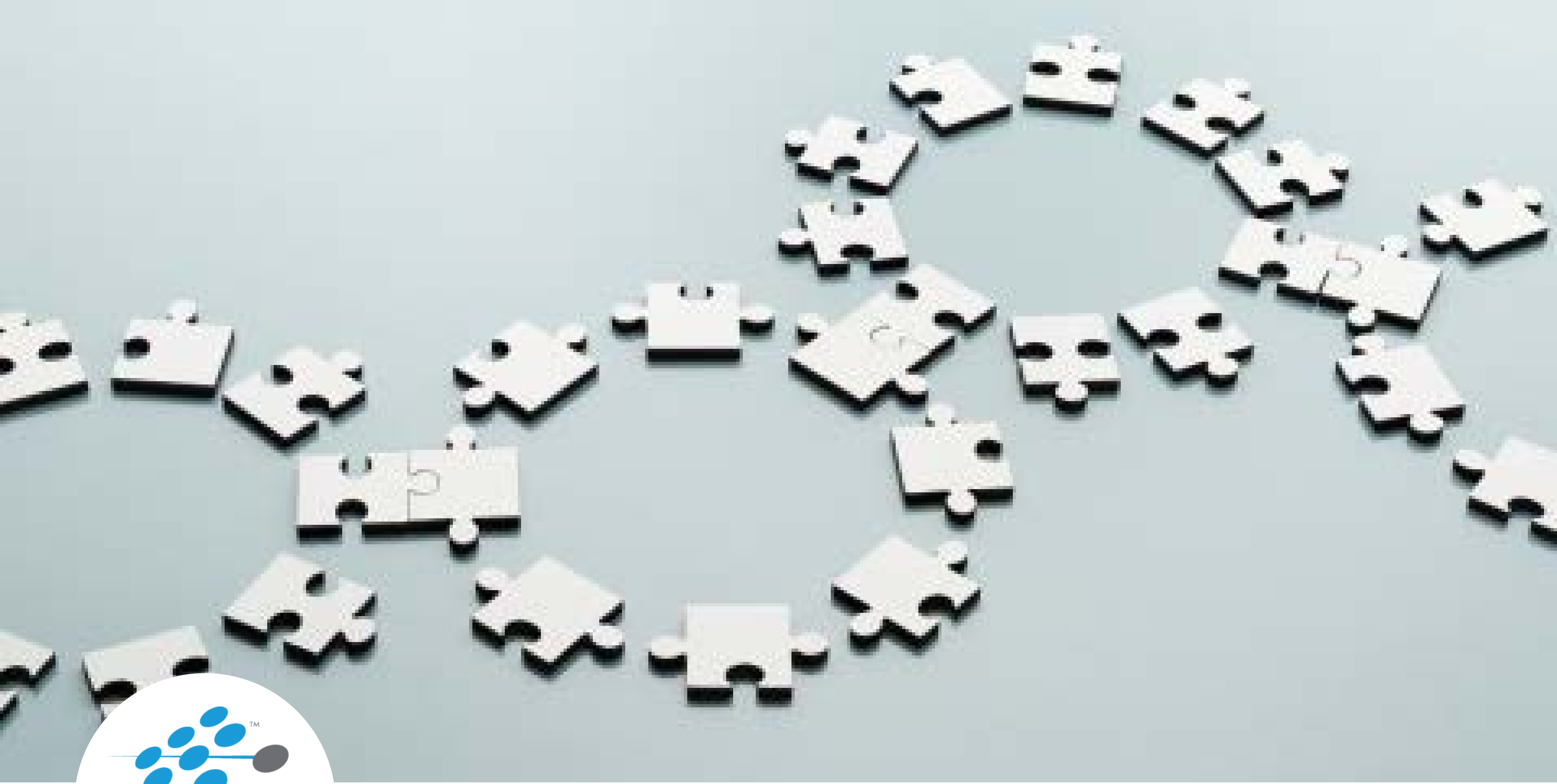
The **SHARED** keyword specifies a definition that is available for use throughout the definition file. Without either SHARED or EXPORT keywords, a definition’s scope is limited to the existing text control.

```
SHARED STRING8 Version := '20021101';
```

Lesson Completed

Proceed to Lesson 16:

Defining Your Data



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 16

Lab Exercise 3 - Defining Your Data

LexisNexis®

Risk Solutions

Exercise 3:

Exercise 3 – Define Your Data (Persons)

Finish what we started in Exercise 1 & 2

1. Open PERSONS.ECL
2. Create RECORD and DATASET
3. Syntax Checking
4. Running a Test Query
5. Output a Recordset
6. Looking at Raw Data

Lab Exercise 3: Requirements

1. Use the definition file that you created in **Lab Exercise 1 as a starting point. This file should be in the *TrainingYourName*** folder and named *Persons.*

2. Create the RECORD definition using the following layouts of the fields. The recommended name of the RECORD definition is **Layout_Persons. Do not EXPORT this definition in this exercise.**

3. The layout of the fields is:

Description	Type Field	Name
Individual Identifier	SIGNED 4-byte integer	RecID
First Name	15-character string	FirstName
Last Name	25-character string	LastName
Middle Name	15-character string	MiddleName
Name Suffix (SR, JR, 1-9)	2-character string	NameSuffix
Date Added YYYYMMDD format	8-character string	FileDate
3-digit numeric code	UNSIGNED 2-byte integer	BureauCode
Marital Status (blank)	1-character string	MaritalStatus
Sex (M, F, N, U)	1-character string	Gender
Number of dependents	UNSIGNED 1-byte integer	DependentCount
Date of Birth YYYYMMDD format	8-character string	BirthDate
Address	42-character string	StreetAddress
City	20-character string	City
State	2-character string	State
5-digit zip code	5-character string	ZipCode

4. Create the DATASET definition, using the name of the file that you sprayed in **Lab Exercise 2**, the RECORD definition that you created above, and determine the type of file defined (THOR, CSV or XML). Make sure to EXPORT this definition.

Lab Exercise 3 RECORD Solution:

```
Layout_Persons := RECORD
  INTEGER4  RecID;
  STRING15  FirstName;
  STRING25  LastName;
  STRING15  MiddleName;
  STRING2   NameSuffix;
  STRING8   FileDate;
  UNSIGNED2 BureauCode;
  STRING1   MaritalStatus;
  STRING1   Gender;
  UNSIGNED1 DependentCount;
  STRING8   BirthDate;
  STRING42  StreetAddress;
  STRING20  City;
  STRING2   State;
  STRING5   ZipCode;
END;
```

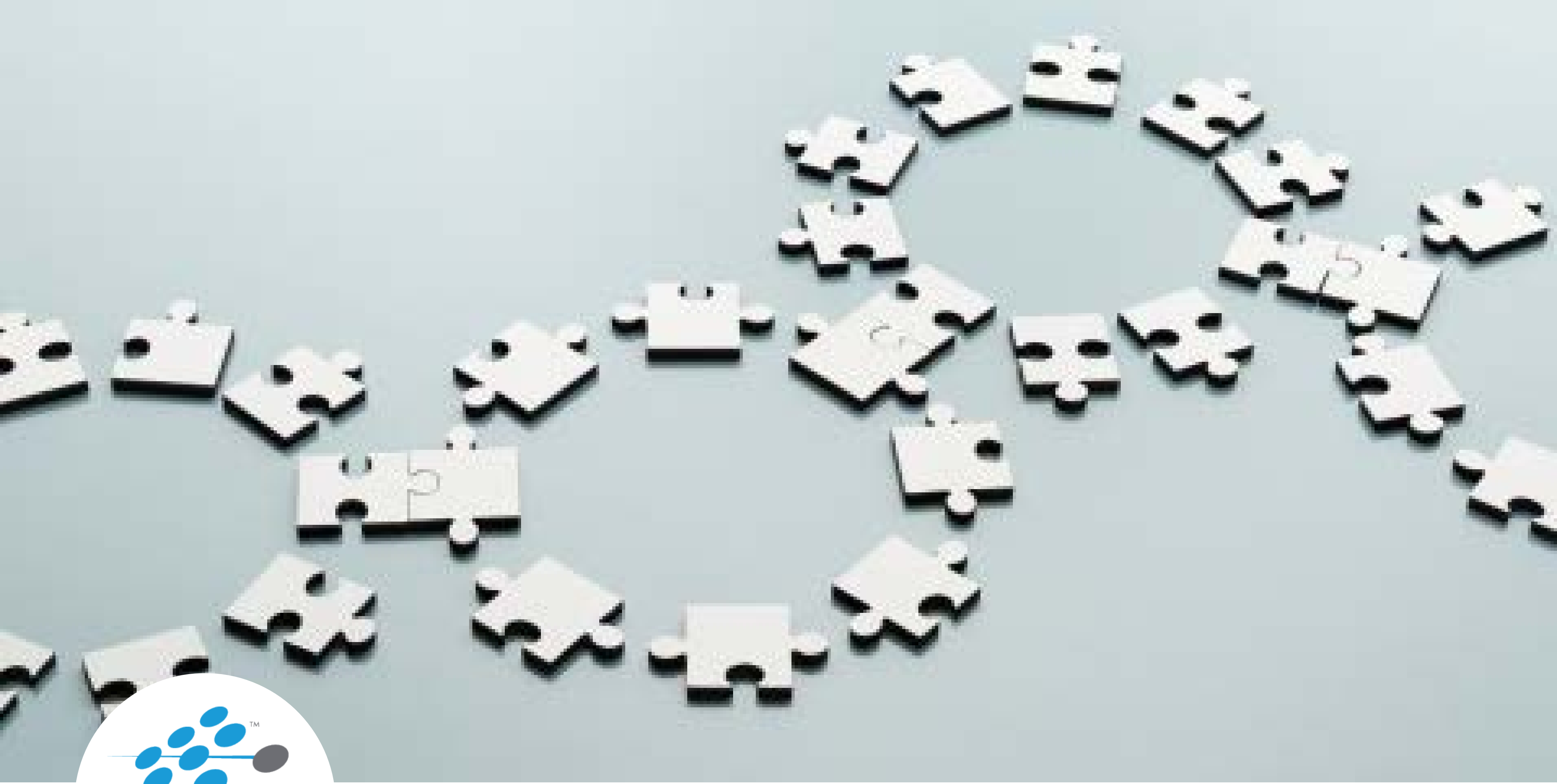
Lab Exercise 3 - Complete Solution:

```
Layout_Persons := RECORD
  INTEGER4  RecID;
  STRING15  FirstName;
  STRING25  LastName;
  STRING15  MiddleName;
  STRING2   NameSuffix;
  STRING8   FileDate;
  UNSIGNED2 BureauCode;
  STRING1   MaritalStatus;
  STRING1   Gender;
  UNSIGNED1 DependentCount;
  STRING8   BirthDate;
  STRING42  StreetAddress;
  STRING20  City;
  STRING2   State;
  STRING5   ZipCode;
END;


EXPORT Persons :=
   DATASET('~ONLINE::XXX::Intro::Persons',Layout_Persons,THOR);
```

Lesson Completed

Proceed to Lesson 17:

OUTPUT and Aggregate Functions



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 17

Using OUTPUT and Aggregate Functions

LexisNexis®

Risk Solutions

Basic Actions

OUTPUT

[*name* :=] **OUTPUT(***recordset* [,*format*] [,*file* [,OVERWRITE]]**)**

name – Optional definition name for this action

recordset – The set of records to process

format – The format of the output records: a previously defined RECORD structure, or an "on-the-fly" record layout enclosed in { } braces.

file – Optional name of file to write the records to. If omitted, formatted data stream returns to the command line or Query Builder program.

OVERWRITE – Allows file to be overwritten if it exists

The **OUTPUT** action writes the *recordset* to the specified *file* in the specified *format*.

OUTPUT Examples:

```
Persons;
OUTPUT(Persons);  //both are equivalent

OUTPUT(Persons,{FirstName,LastName}, NAMED('Names_Only'));

OUTPUT(BHF_Out,,'OUT::Business_Header_' +
                  Business_Header.Version, OVERWRITE);

OUTPUT(SORT(T_Persons,ZipCode), MyTFormat, 'ziplist::fred.out');

OUTPUT(COUNT(Persons(State = 'FL'));  //scalar value output
```

Aggregate Functions - COUNT

COUNT(*recordset***)**

COUNT(*valuelist***)**

recordset – The set or set of records to process.

valuelist - A comma-delimited list of expressions to count. This may also be a SET of values.

The **COUNT** function returns the number of records in the specified *recordset*.

```
COUNT(Person(per_state IN ['FL','NY']));
TradeCount := COUNT(Trades);
```

Aggregate Functions - MAX

MAX(*recordset , value* **)**

MAX(*valuelist***)**

recordset – The set or set of records to process.

value – The field or expression to find the maximum value of.

valuelist - A comma-delimited list of expressions to find the maximum value of. This may also be a SET of values

The **MAX** function returns the maximum value of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

MaxBal := **MAX**(Trades, Trades.trd_bal);

Aggregate Functions - MIN

MIN(*recordset , value* **)**

MIN(*valuelist***)**

recordset – The set or set of records to process.

value – The field or expression to find the minimum value of.

valuelist - A comma-delimited list of expressions to find the minimum value of. This may also be a SET of values

The **MIN** function returns the minimum value of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

```
MinBal := MIN(Trades, Trades.trd_bal);
```

Aggregate Functions - SUM

SUM(*recordset* **,** *value* **)**

SUM(*valuelist***)**

recordset – The set or set of records to process.

value – The expression or field in the *recordset* to sum.

valuelist - A comma-delimited list of expressions to find the sum of. This may also be a SET of values.

The **SUM** function returns the additive sum of the value contained in the specified *field* for each record of the *recordset*. Returns 0 if the *recordset* is empty.

SumBal := **SUM**(Trades, Trades.trd_bal);

Aggregate Functions - AVE

AVE(*recordset* , *value* **)**

AVE(*valuelist***)**

recordset – The set or set of records to process.

value – The field or expression to find the average value of.

valuelist - A comma-delimited list of expressions to find the average of. This may also be a SET of values.

The **AVE** function returns the average value (arithmetic mean) of the specified *field* from the specified *recordset*. Returns 0 if the *recordset* is empty.

```
AvgBal := AVE(Trades, Trades.trd_bal);
```

Builder Window Runnable Files (BWR)

- ✓ Stored in Repository – Ready to Run!
- ✓ Rules:
  - ✓ File must have at least one action
  - ✓ No EXPORT or SHARED
  - ✓ References to other definitions must be fully qualified:

```
IMPORT $;
$.Persons;
COUNT($.Persons);
OUTPUT($.Persons,{ID,LastName,FirstName});
OUTPUT($.Persons,{StreetAddress,City,State,ZipCode},NAMED('Address_Info');
```

Lab Exercise Overview:

Exercise 4 – Basic Queries

- Use of IMPORT
- Use of COUNT
- Use of OUTPUT
- Output a Recordset
- Looking at Raw Data

Lab Exercise 4:

Exercise Spec:

Explore some very basic ways to query your data. Normally these queries are performed just after a spray and RECORD and DATASET defines, in order to verify that they are correct and the data looks reasonable. The first rule of ECL is to *know your data*!

Steps:

1. Open a new ECL definition file in your repository, naming it **BWR_BasicQueries**.
2. Comment out or delete the starting line in the **BWR_BasicQueries** file that you just created:
// export BWR_BasicQueries := 'todo';
3. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
4. Generate an output for the *Persons* table by simply using the name of the definition in your ECL file.
5. Generate a count of all records in the *Persons* table.
6. Generate an output for the *Persons* table, limiting the output to the *RecID*, *LastName*, and *First Name*.
7. Generate an output for the *Persons* table, limiting the output to the *RecID*, *StreetAddress*, *City*, *State* and *ZipCode*, and name the output tab in the ECL IDE "*Address_Info*" (Hint: review the Named OUTPUT section in this lesson).

Result Comparison:

Verify in the ECL IDE Results tab that all outputs look reasonable.

The count for the Persons table should be 963512.

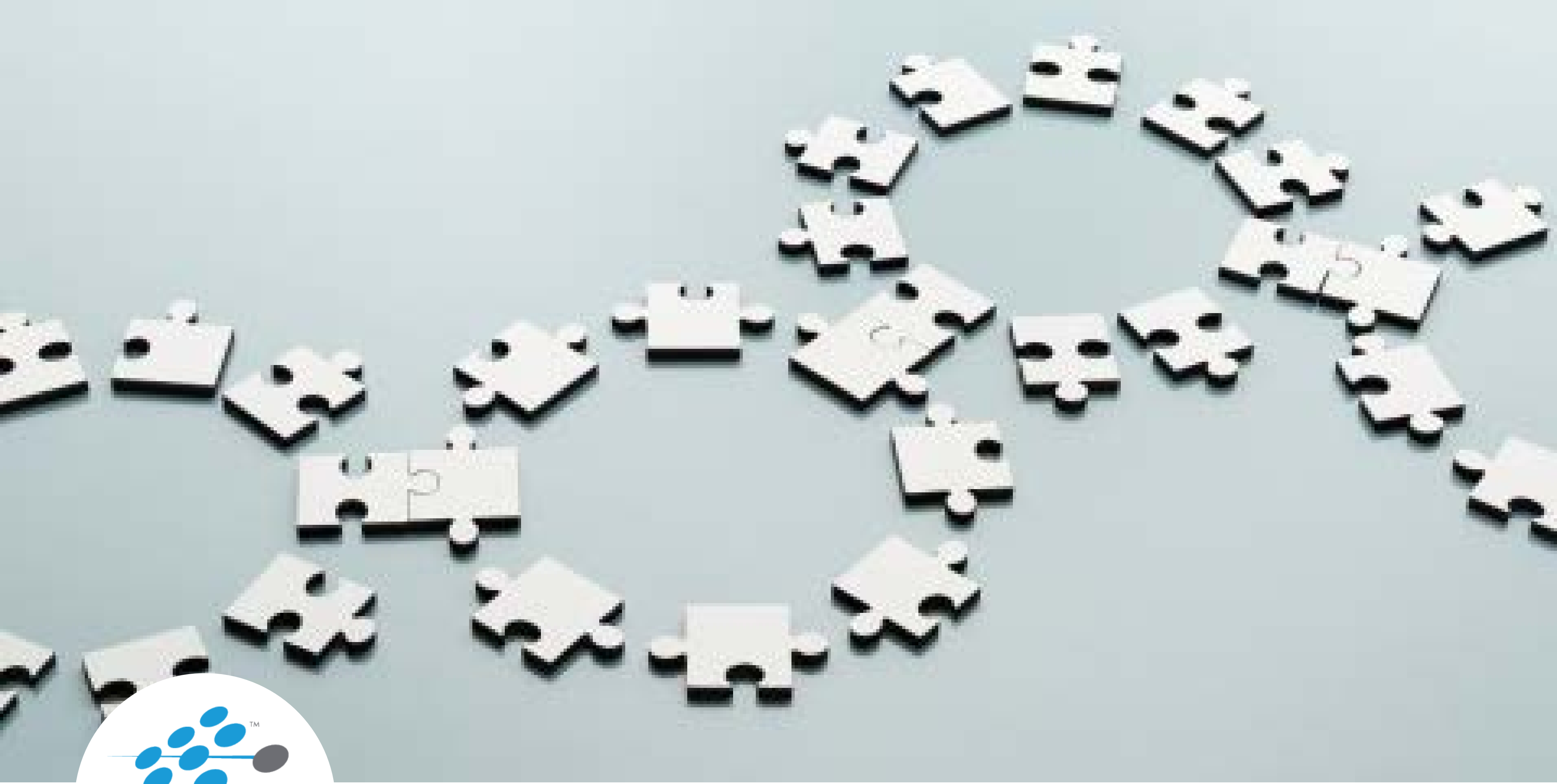
Lab Exercise 4 Solution:

```
//EXPORT BWR_BasicQueries := 'todo';
IMPORT $;
$.Persons;
COUNT($.Persons);
OUTPUT($.Persons,{RecId,LastName,FirstName});
OUTPUT($.Persons,{RecId,StreetAddress,City,State,ZipCode},NAMED('Address_Info'));
```

Lesson Completed

Proceed to Lesson 18:

Filtering your Data



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 18

Lab Exercise 5 – Filtering Your Data

Risk Solutions

Recordset Filters - Review

- ✓ Any Boolean expression within parentheses following a Dataset or Recordset name is a **filter expression**, used to define the specific subset of records to use
- ✓ Multiple filter conditions may be specified by separating each filter expression with a comma (,) or explicitly using the AND operator

T_Names := People(**Lastname >= 'T', Lastname < 'U'**);

RateGE7Trds := Trades(**trd_rate >= 7**);

ValidTrades := Trades(**NOT IsMortgage AND NOT IsClosed**);

Set Ordering and Indexing (cond)

✓ *Strings (character sets)* - may also be indexed to access individual characters within the string

```
MyString   := 'ABCDE';
MiddleChar := MyString[3];     //MiddleChar contains "C"
```

✓ *Substrings* - may be extracted by using 2 periods to separate the beginning and ending element numbers to specify the substring to extract

```
MySubString1 := MyString[2..4];  // 'BCD'
MySubString2 := MyString[..4];   // 'ABCD'
MySubString3 := MyString[2..];   // 'BCDE'
```

Logical Operators:

Logical Operators

NOT	Boolean NOT operation
~(tilde)	Boolean NOT operation
AND	Boolean AND operation
OR	Boolean OR operation

Lab Exercise Overview:

Exercise 5 – Filters

- Basic Filters
- More use of COUNT
- Use of String Indexing
- Use of Logical Operators

Lab Exercise 5:

Exercise Spec:

Create some simple filters on the persons dataset that provides meaningful information and analysis. Except for the file specified below, do not use any new definitions in this exercise; use the existing **Persons** ECL definition file to generate meaningful output in the ECL IDE.

Steps:

1. Create a new definition file and name it **BWR_BasicPersonsFilters**. Use this file to create and generate all of your queries in this exercise.
2. Comment out or delete the starting line in the**BWR_BasicPersonsFilters** file that you just created:

```
// export BWR_BasicPersonsFilters := 'todo';
```

3. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
4. Filter and Count all persons who live in the state of Florida. Your expected count is 46887.
5. Filter and Count all persons who live in the state of Florida and the city of Miami. Your expected count is now 2821.
6. Filter and Count all persons who live in the zip code of 33102. Your expected count is now 53.
7. Filter and Count all persons whose First Name begins with the letter 'B'. Your expected count is 35619.
8. Filter and Count all persons whose file date year is after 2000. Your expected count is 785.

Result Comparison:

The results for each step reflected by the expected COUNTs are shown in the steps above.

Lab Exercise 5 Solution:

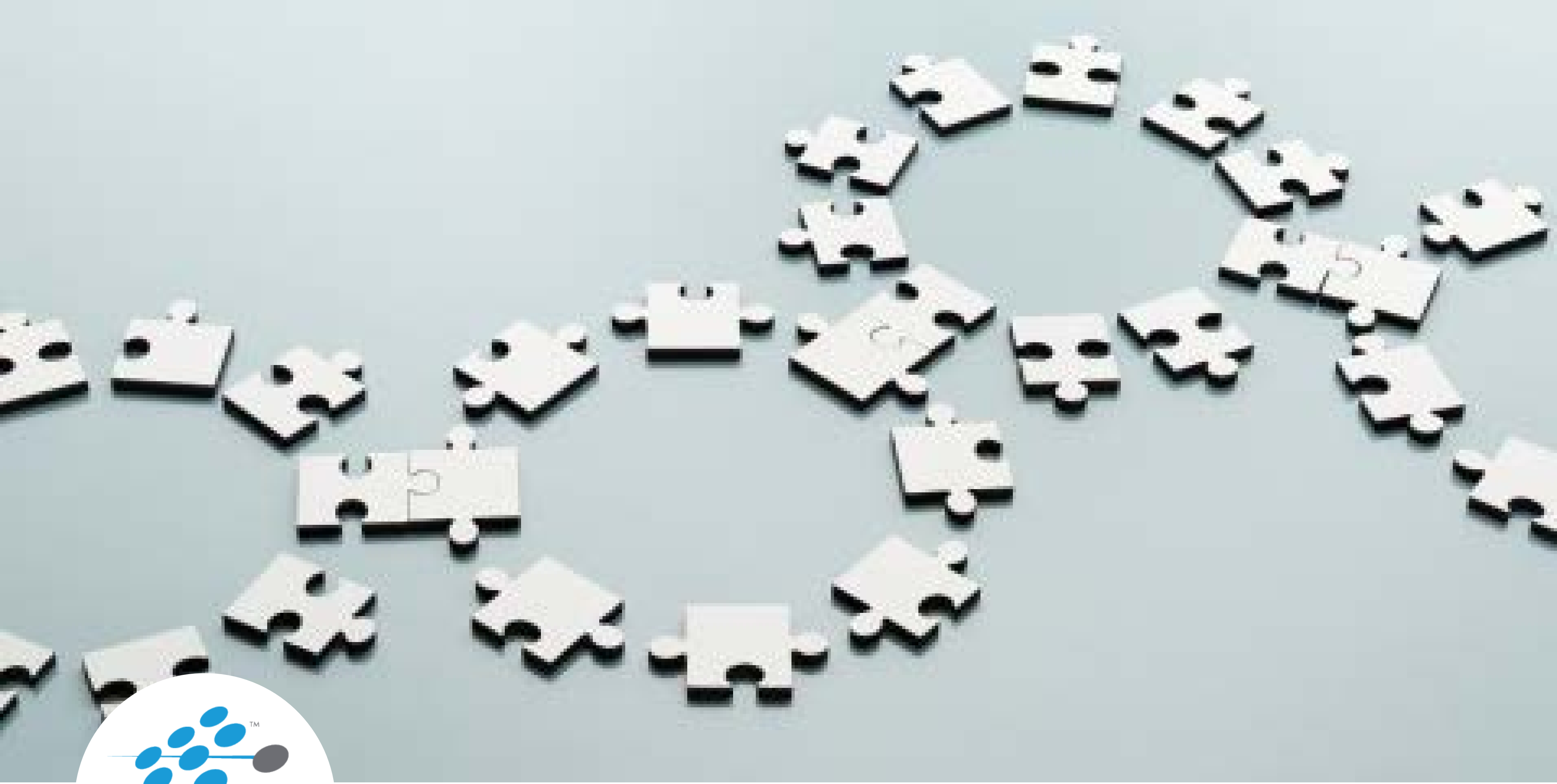
```
//EXPORT BWR_BasicPersonsFilters := 'todo';
IMPORT $;

COUNT($.Persons(State='FL'));
COUNT($.Persons(State='FL',City='MIAMI'));
COUNT($.Persons(ZipCode='33102'));

COUNT($.Persons(FirstName[1]='B'));
COUNT($.Persons(FileDate[1..4] > '2000'));
```

Lesson Completed

Proceed to Lesson 19:

ECL Best Practices &

Boolean Definitions



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 19

ECL Definition Creation & Lab Exercise 6 – Boolean Definitions

LexisNexis®

Risk Solutions

ECL Best Practices

- Comparison to other languages
- Atomic Programming
- Growing Solutions
- Ugly ECL
- Easy Optimization

ECL Best Practices – Definition Creation

Comparison to Other Languages

The similarities come from the fundamental requirement of solving any data processing problem: First, understand the problem. After that, in many programming environments, you use a “top-down” approach to map out the most direct line of logic to get your input transformed into the desired output.

This is where the process diverges in ECL, because the tool itself requires a different way of thinking about forming the solution, and the direct route is not always the fastest. ECL requires a “bottom-up” approach to problem solving.

ECL Best Practices – Definition Creation

"Atomic" Programming

In ECL, once you understand what the end result needs to be, you ignore the direct line from problem input to end result and instead start by breaking the issue into as many pieces as possible—the smaller the better. By creating "atomic" bits of ECL code, you've done all the known and easy bits of the problem. This usually gets you 80% of the way to the solution without having to do anything terribly difficult.

Once you've taken all the bits and made them as atomic as possible, you can then start combining them to go the other 20% of the way to produce your final solution. In other words, you start by doing the little bits that you know you can easily do, then use those bits in combination to produce increasingly complex logic that builds the solution organically.

ECL Best Practices – Definition Creation

Growing Solutions

The basic ECL Definition building blocks are the Set, Boolean, Recordset, and Value Definition types. Each of these can be made as "atomic" as needed so that they may be used in combination to produce "molecules" of more complex logic that may then be further combined to produce a complete "organism" that produces the final result.

For example, assume you have a problem that requires you to produce a set of records wherein a particular field in your output must contain one of several specified values (say, 1, 3, 4, or 7). In many programming languages, the pseudo-code to produce that output would look something like this:

```
Start at top of MyFile
Loop through MyFile records
If MyField = 1 or MyField = 3 or MyField = 4 or MyField = 7
Include record in output set
Else
Throw out record and go back to top of loop
end if and loop
```

ECL Best Practices – Definition Creation

Growing Solutions (Cont)

While in ECL, the actual code would be:

```
SetValidValues := [1,3,4,7];                    //Set Definition
IsValidRec := MyFile.MyField IN SetValidValues; //Boolean
ValRecsMyFile := MyFile(IsValidRec);            //filtered Recordset
OUTPUT(ValRecsMyFile);
```

The thought process behind writing this code is:

"I know I have a set of constant values in the spec, so I can start by creating a Set definition of the valid values...“

"And now that I have a Set defined, I can use that Set to create a Boolean definition to test whether the field I‘m interested in contains one of the valid values..."

"And now that I have a Boolean defined, I can use that Boolean as the filter condition to produce the Recordset I need to output."

The process starts with creating the Set definition "atom," then using it to build the Boolean "molecule," then using the Boolean to grow the "organism" - the final solution.

ECL Best Practices – Definition Creation

"Ugly" ECL is Possible, Too

Of course, that particular set of ECL could have been written like this (following a more top down thinking process):

```
OUTPUT(MyFile(MyField IN [1,3,4,7]));
```

The end result, in this case, would be the same.

However, the overall usefulness of this code is drastically reduced because, in the first form, all the “atomic” bits are available for re-use elsewhere when similar problems come along. In this second form, they are not. And in all programming styles, code re-use is considered to be a good thing.

ECL Best Practices – Definition Creation

Easy Optimization:

By breaking your ECL code into its smallest possible components, you allow ECL's optimizing compiler to do the best job possible of determining how to accomplish your desired result. This leads to another dichotomy between ECL and other programming languages: usually, the less code you write, the more "elegant" the solution; but in ECL, the more code you write, the better and more elegant the solution is generated for you. Remember, your definitions tell the compiler what to do, not how to do it.

The more you break down the problem into its component pieces, the more leeway you give the optimizer to produce the fastest, most efficient executable code.

ECL Definition Types - Boolean

Boolean –

A Boolean definition is a logical expression resulting in a TRUE/FALSE result

```
IsGoodClass      := TRUE;
IsFloridian      := People. state = 'FL';
IsSeniorCitizen  := People.Age >= 65;
```

Lab Exercise:

Exercise 6:

- Boolean Definitions

- Review Naming Conventions
- IMPORT and EXPORT needed

Lab Exercise 6:

Exercise Spec:

Create a Boolean Attribute that will be TRUE for all male Persons living in Florida who were born after the year 1979. This will be used in subsequent exercises to filter the Persons dataset.

Steps:

1. Create a new EXPORT Boolean definition file called **IsYoungFloridaMale**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. Create a local Boolean definition (remember that local means that the definition is not added separately to the repository-it will be contained inside the **IsYoungFloridaMale** file, neither EXPORT nor SHARED) called **IsFloridian** that will test if a person lives in Florida. Check your Persons record definition to determine the name of the state field. The abbreviation for Florida is 'FL'
4. Create another local Boolean definition called **IsMale** that will test if a person's gender is marked as male ('M'). Check your Persons record definition to determine the name of the gender field.
5. Create a third local Boolean definition called **IsBorn80** that will test if a person was born after the year 1979. Check your Persons record definition to determine the name of the birth date field. Use string indexing to check for birth dates after the year 1979. Make sure to also eliminate records with no birth dates.
6. Create the **IsYoungFloridaMale** Boolean definition so that it results in TRUE for any male living in Florida and born after the year 1979. Use the three local Boolean definitions (**IsFloridian**, **IsMale**, **IsBorn80**) you just created.

Result Comparison:

At this time you only need to make sure that your ECL syntax is correct. We will use this attribute in a subsequent exercise to verify its accuracy.

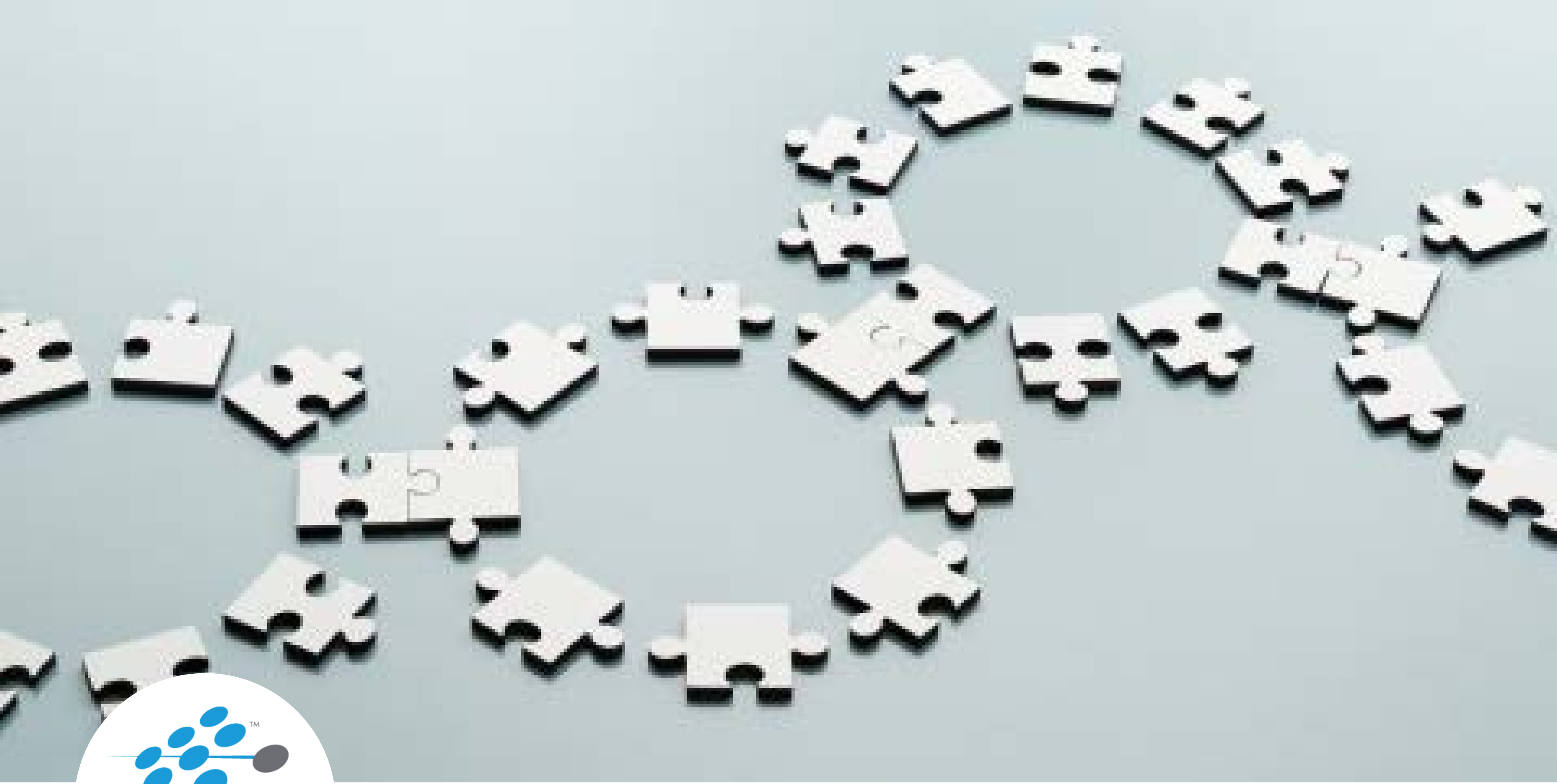
Lab Exercise 6 Solution:

```
IMPORT $;
IsFloridian := $.Persons.State = 'FL';
IsMale      := $.Persons.Gender = 'M';
IsBorn80    := $.Persons.Birthdate <> '' AND $.Persons.Birthdate[..4] >= '1980' ;

EXPORT IsYoungFloridaMale := IsFloridian AND
                             IsMale AND
                             IsBorn80;
```

Lesson Completed

Proceed to Lesson 20:

Set Definitions



HPCC SYSTEMS®

HPCC Systems:
Introduction to the Enterprise Control Language

Lesson 20

Lab Exercise 7 – Set Definitions

LexisNexis®

Risk Solutions

ECL Definition Types - Set

Set –

A Set definition is a set of explicitly declared constant values or expressions within square brackets (all elements must be the same type)

```
SetTrueFalseValues    := [0, 1];
SetSoutheastStates    := ['FL','GA','AL','SC'];
SetStatusCodes        := ['1','X','9','W'];
SetInts               := [1,2+3,45,def1,7*3,def2];
```

Dynamic Sets: SET Function

SET(*recordset, field* **)**

recordset – The set of records from which to derive the SET of values.

field – The field in the *recordset* from which to obtain the values.

The **SET** function returns a SET for use in any set operation (such as the IN operator).

```
r := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'},{'F'},{'G'},{'H'},{'I'},{'J'}],r);
SetLettersD_J := SET(SomeFile(Letter > 'C'), Letter);
y := 'A' IN SetLettersD_J;          //results in FALSE
z := 'D' IN SetLettersD_J;          //results in TRUE
//
MyZipSet := SET($.Persons, ZipCode);
```

Lab Exercise:

Exercise 7:

- SET Definitions

- Review Naming Conventions
- IMPORT and EXPORT needed
- Defining SETs
- SET function

Lab Exercise 7:

Exercise Spec:

A Set definition is any whose expression is a set of values, defined within square brackets. Create a static and dynamic set definition for the Persons table.

Steps:

1. Create an EXPORT ECL definition called **SetMStates**.
2. Import all definitions from your target Training Module. (TrainingYourName).
3. Use the following set of strings to create the **SetMStates** definition:

```
MA, MD, ME, MI, MN, MP, MO, MS, MT
```

4. Emulating the three steps above, create another EXPORT ECL definition called **SetBureauCodes** (new ECL file), but use the SET function to create a dynamic set definition of all bureau codes found in the Persons dataset.

Result Comparison:

Make sure that the syntax check is correct for all ECL definitions that you created. We will verify if they are logically correct in a subsequent exercise.

Lab Exercise 7 Solution:

SetMStates.ECL:

```
EXPORT SETMStates := ['MA','MD','ME','MI','MN','MO','MP','MS','MT'];
```

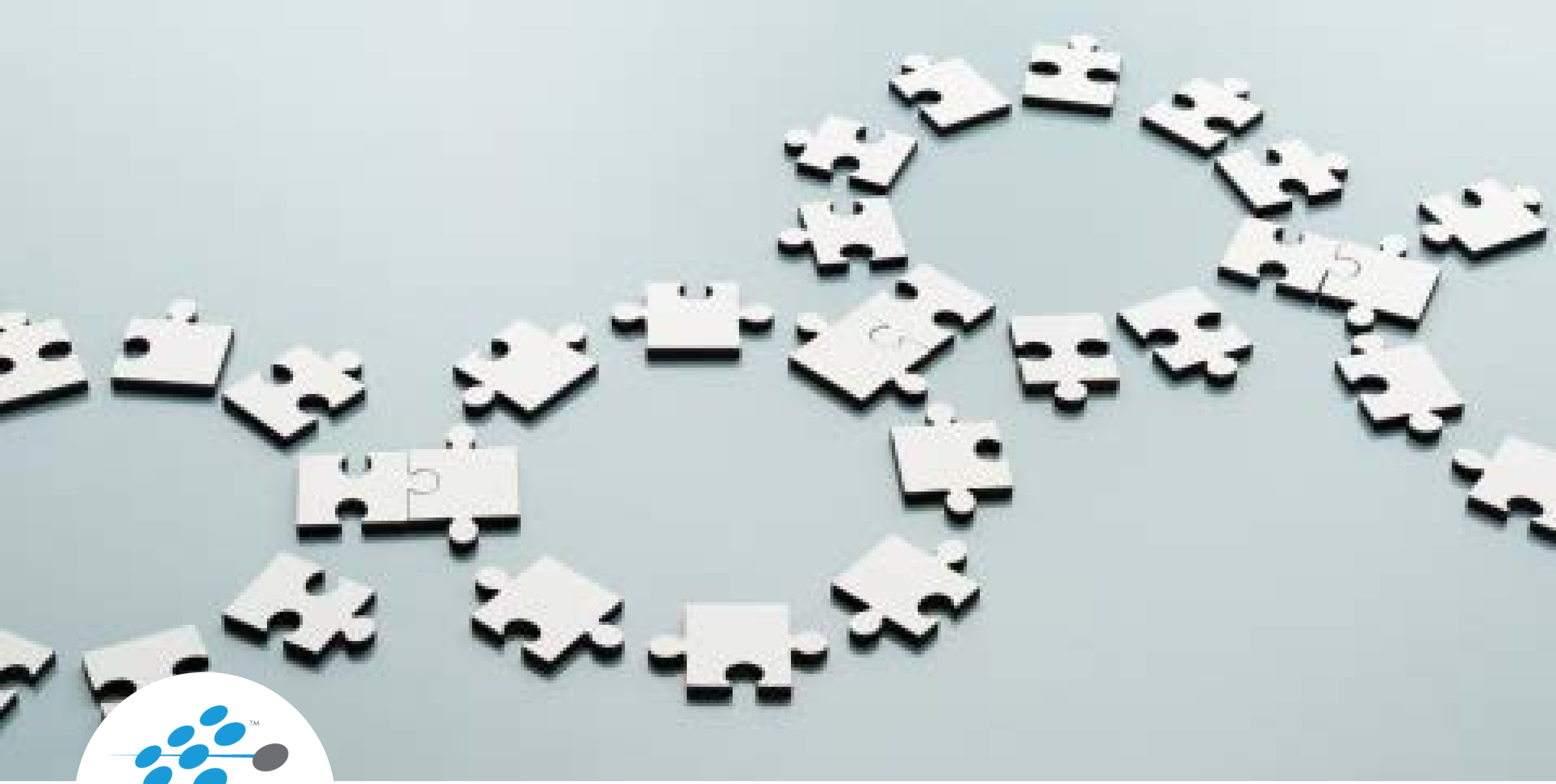
SetBureauCodes.ECL:

```
IMPORT $;
EXPORT SetBureauCodes := SET($.Persons,BureauCode);
```

Lesson Completed

Proceed to Lesson 21:

Recordset Definitions



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 21

Lab Exercise 8 – Recordset Definitions

Risk Solutions

ECL Definition Types - Recordset

Recordset –

A Recordset definition is a filtered dataset or recordset

```
FloridaPeople          := People(IsFloridian);
SeniorFloridaPeople    := FloridaPeople(IsSeniorCitizen);
```

Recordset Ordering and Indexing

- ✓ *Recordsets* - may also be indexed to access individual records (or ranges) within the recordset or dataset

```
MySortedRecs := SORT(People(LastName = 'TAYLOR'), FirstName);

FirstTaylor   := MySortedRecs[1];      //First record in the sorted set
FirstTaylors := MySortedRecs[1..3];  //First three records
```

- ✓ *Fields in Records* - may be accessed using indexing to access the individual record then qualifying the field to return

```
FirstTaylorFirstName := MySortedRecs[1].FirstName;
                //First name in first record in the sorted set
```

Recordset Filters

Any Boolean expression within parentheses following a Dataset or Recordset name is a **filter expression**, used to define the specific subset of records to use

Multiple filter conditions may be specified by separating each filter expression with a comma (,) or explicitly using the AND operator

```
T_Names := People(Lastname >= 'T', Lastname < 'U');

RateGE7Trds := Trades(trd_rate >= 7);

ValidTrades := Trades(NOT IsMortgage AND NOT IsClosed);
```

Expressions: IN Operator

IN Operator

value **IN** *value_set*

value – The value definition or constant to search for in the *value_set*.

value_set – The set to search.

The **IN** operator is a shorthand for a collection of OR conditions. It searches the *value_set* to find a match for the *value* and returns a Boolean TRUE or FALSE.

```
SetSoutheastStates := ['FL','GA','AL','SC'];
BOOLEAN IsSoutheastState(STRING2 state) :=  state IN SetSoutheastStates;
```

Lab Exercise 8:

Exercise 8

- ✓RecordSET Definitions

- ✓Reuse of BOOLEAN and SET definitions
- ✓IMPORT and EXPORT needed
- ✓Filtering

Lab Exercise 8:

Exercise Spec:

Create a RecordSet definition for the set of male persons living in Florida who were born after 1979, and create another a RecordSet definition for the set of male persons living in US states beginning with 'M' .

Steps:

1. Create an EXPORT RecordSet definition called **YoungMaleFloridaPersons**.
2. Create an EXPORT RecordSet definition called **MeninMStatesPersons**.
3. In both files above, IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
4. For the **YoungMaleFloridaPersons** definition, use the **IsYoungFloridaMale** Boolean definition that you created in Exercise 6.
5. For the **MeninMStatesPersons** definition, use the **SetMStates** Set definition that you created in Exercise 7.

Result Comparison:

Use a new Builder window and verify that the **YoungMaleFloridaPersons** output data looks correct. Also, perform a COUNT and verify that the result is 529.

Open a new Builder window and verify that the **MenInMStatesPersons** output data looks correct. Also, perform a COUNT and verify that the result is 65388.

Lab Exercise 8 Solutions:

YoungMaleFloridaPersons.ECL:

```
IMPORT $;
EXPORT YoungMaleFloridaPersons := $.Persons($.IsYoungFloridaMale);
```

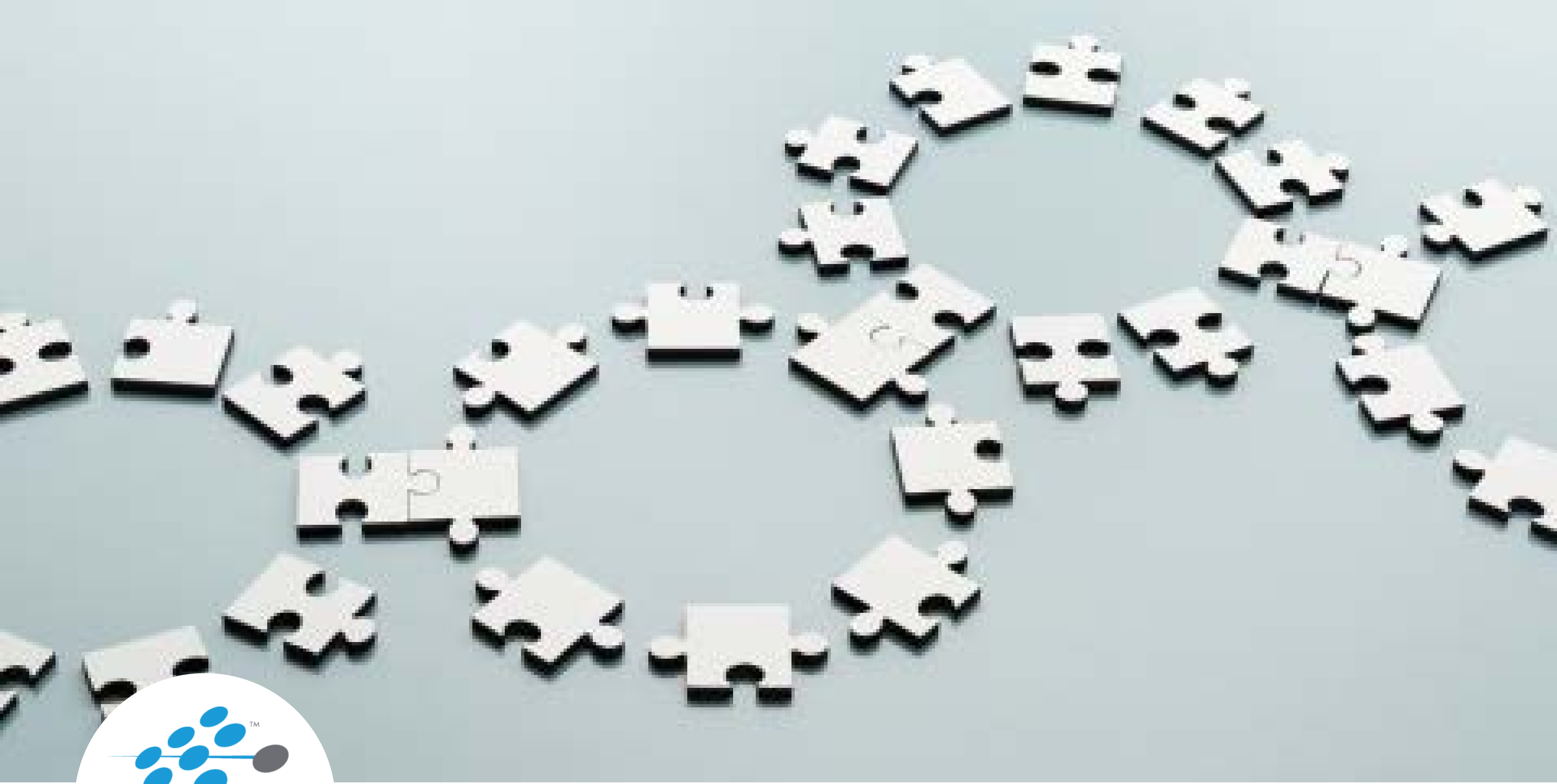
MenInMStates.ECL:

```
IMPORT $;
EXPORT MenInMStatesPersons := $.Persons(State IN $.SetMStates,
                                        Gender = 'M');
```

Lesson Completed

Proceed to Lesson 22:

Conditional Functions



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 22

Conditional ECL Functions

LexisNexis®

Risk Solutions

Conditional Functions - IF

IF(*expression, truevalue, falsevalue***)**

expression – A conditional expression.

truevalue – If the *expression* is true, this is the value or recordset to return, or an action to perform.

falsevalue – If the *expression* is false, this is the value or recordset to return, or an action to perform.

```
TradeDate := IF(trades.status IN ['A','D','0','1'],
                trades.lastdate, '');
```

Conditional Functions - MAP

MAP(*condition=>val*,[*condition=>val*,… ,] *elseval*]**)**

- ✓ *condition* – A conditional expression.
- ✓ => The "results in" operator. Valid only in CASE, MAP and CHOOSETS.
- ✓ *val* – the value to return if the *condition* is true. This may be a single value, recordset, or action.
- ✓ *elseval* – The value to return if all *conditions* are false. This may be a single value, recordset (optional), or action (optional).

Conditional Functions – MAP Example

```
Valu6012 := MAP(
          NOT EXISTS(Trades)                => 99,
          NOT EXISTS(Trades(isValid))       => 98,
          NOT EXISTS(Trades(isValidDate))   => 96,
          Count6012);
   /* If there are no trades, Valu6012 gets 99
      else if there are no valid trades, Valu6012 gets 98
      else if there are no valid dates, Valu6012 gets 96
      else, Valu6012 gets Count6012
   */
```

Conditional Functions - CASE

CASE(*expression, caseval* => *val*,[...] *elseval***)**

- ✓ *expression* – An expression that results in a single value.
- ✓ *caseval* – A value to compare against the result of the expression.
- ✓ => The "results in" operator.
- ✓ *val* – The value to return - a single value, recordset, or action.
- ✓ *elseval* – The value to return if the result of the *expression* does not match any of the *caseval* values. This may be a single value, recordset , or action.

```
NumString123 := CASE(num,1 => 'one',2 => 'two',3 => 'three', 'error');
```

Conditional Functions - CHOOSE

CHOOSE(*expression*, *value*,[…, *value*,] *elseval***)**

- ✓ *expression* – An arithmetic expression that results in a positive integer and determines which *value* parameter to return.
- ✓ *value* – the values to return. There may be as many *value* parameters as necessary to specify all the expected values of the *expression.*
- ✓ *elseval* – The value to return when the *expression* returns an out-of-range value. The last parameter is always the *elseval*.

```
NumString123 := CHOOSE(num,'one','two','three', 'error');
```

Conditional Functions – REJECTED/WHICH

REJECTED(*condition,…,condition***)**

condition – A conditional expression to evaluate.

```
RejectCode := REJECTED(Person.income > 20000,
                       Person.age < 80,
                       Person.state <> 'NY');
```

WHICH(*condition,…,condition***)**

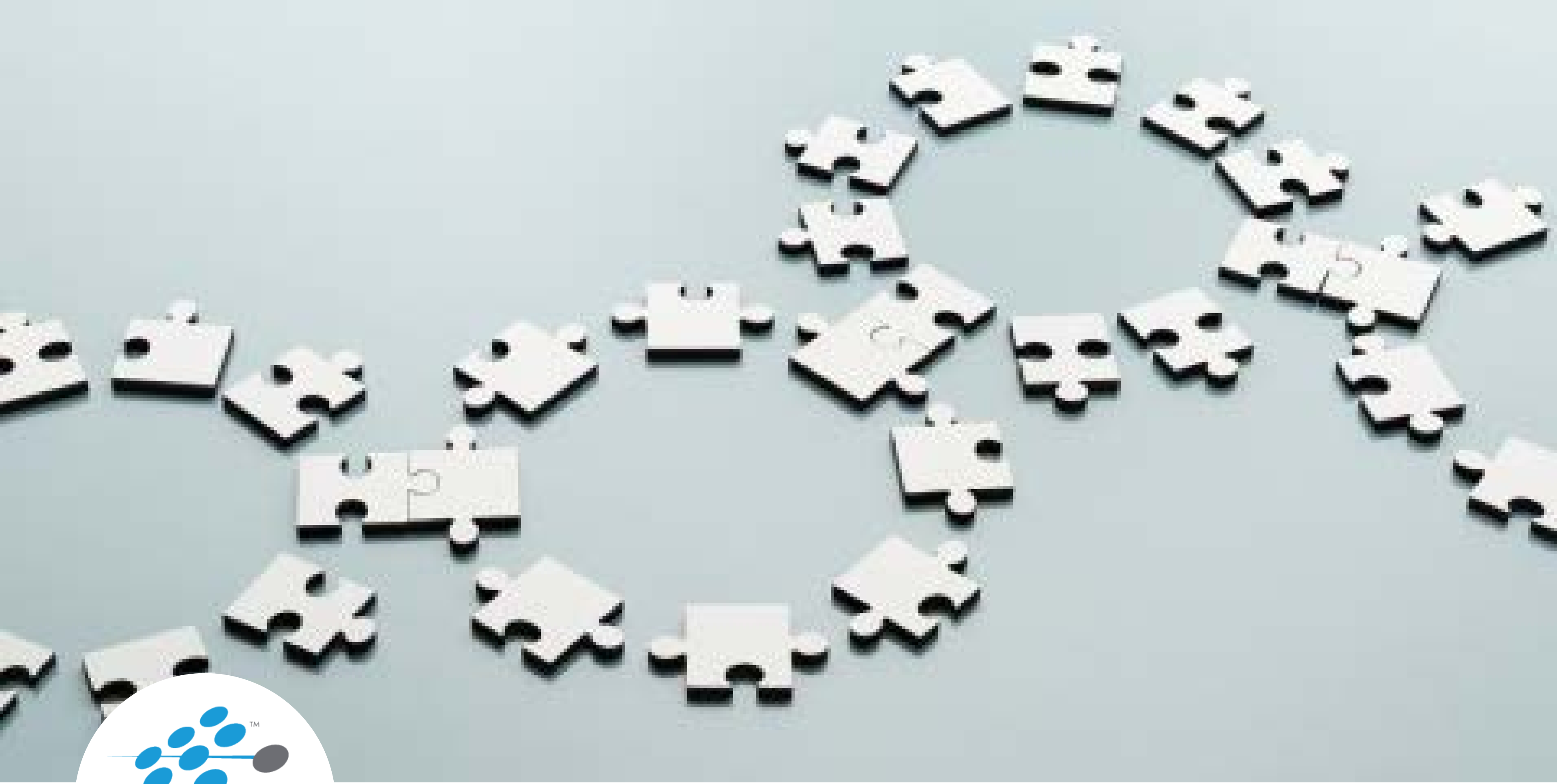
condition – A conditional expression to evaluate.

```
AcceptCode := WHICH(Person.state IN SetSoutheastStates,
                    Person.state IN SetMidwestStates,
                    Person.state IN SetNorthwestStates);
```

Lesson Completed

Proceed to Lesson 23:

Mathematical Functions



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 23

Mathematical Functions

LexisNexis®

Risk Solutions

Mathematical Functions – ROUND

ROUND(*realvalue***)**

✓ *realvalue* – The floating-point value to round.

```
Amount := 1000.49;
RoundAmount := ROUND(Amount);
                       // Rounded amount is 1000;
```

Mathematical Functions – ABS and ACOS

ABS(*expression***)**

✓ *expression* – The value (REAL or INTEGER) for which to return the absolute value.

```
CreditValue    := ABS(credit_amount);
AbsVal1        := ABS(-1);  // returns 1
```

ACOS(*cosine***)**

✓ *cosine* – The REAL cosine value for which to find the arccosine.

```
ArcCosine := ACOS(CosineAngle);
```

ASIN, ATAN, and ATAN2

ASIN(*sine*)

- ✓ *sine* – The REAL sine value for which to find the arcsine.

```
ArcSine := ASIN(SineAngle);
```

ATAN(*tangent*)

- ✓ *tangent* – The REAL tangent value for which to find the arctangent.

```
ArcTangent := ATAN(TangentAngle);
```

ATAN2(*y,x*)

- ✓ *y,x* – The REAL numerator and denominator values for which to find the arctangent.

```
ArcTangent := ATAN2(x,y);
```

COS and COSH

COS(*angle*)

- ✓ *angle* – The REAL radian value for which to find the cosine.

```
Rad2Deg      := 57.295779513082;  // # degrees in a radian
Deg2Rad      := 0.0174532925199;  // # radians in a degree
Angle45      := 45 * Deg2Rad;     // 45 degrees in radians
Cosine45     := COS(Angle45);     // Cosine of 45 degree angle
```

COSH(*angle*)

- ✓ *angle* – The REAL radian value for which to find the hyperbolic cosine.

```
HypCosine45 := COSH(Angle45);
```

EXP, LN, and LOG

EXP(*n*)

- ✓ *n* – The REAL number to evaluate.

Interim := ROUND(1000 * (**EXP**(ThisSum)/(1+**EXP**(ThisSum))));

LN(*n*)

- ✓ *n* – The REAL number to evaluate.

LnPI := **LN**(3.14159);

LOG(*n*)

- ✓ *n* – The REAL number to evaluate.

LogPI := **LOG**(3.14159);

POWER, RANDOM, and ROUNDUP

POWER(*base, exponent*)

- ✓ *base* – The REAL number to raise.
- ✓ *exponent* – The REAL power to raise base to.

```
Cube2 := POWER(2.0,3.0);
```

RANDOM()

```
Pick100 := RANDOM() DIV 100;
```

ROUNDUP(*realvalue*)

- ✓ *realvalue* – The floating-point value to round.

```
Val := 3.14159;
ValUp := ROUNDUP(Val);  // Result is 4
```

SIN and SINH

SIN(*angle*)

- ✓ *angle* – The REAL radian value for which to find the sine.

```
Rad2Deg    := 57.295779513082;  // # degrees in a radian
Deg2Rad    := 0.0174532925199;  // # radians in a degree
Angle45    := 45 * Deg2Rad;     // 45 degrees in radians
Sine45     := SIN(Angle45);     // Sine of 45 degree angle
```

SINH(*angle*)

- ✓ *angle* – The REAL radian value for which to find the hyperbolic sine.

```
HypSine45 := SINH(Angle45);
```

SQRT and TAN

SQRT(*n*)

- ✓ *n* – The REAL number to evaluate.

```
Root16 := SQRT(16.0);  // Result is 4.0
```

TAN(*angle*)

- ✓ *angle* – A REAL radian value.

```
Rad2Deg     := 57.295779513082;    // # degrees in a radian
Deg2Rad     := 0.0174532925199;    // # radians in a degree
Angle45     := 45 * Deg2Rad;       // 45 degrees in radians
Tangent45   := TAN(Angle45);       // Tangent of 45 degree angle
```

TANH and TRUNCATE

TANH(*angle*)

- ✓ *angle* – A REAL radian value.

```
HypTan45 := TANH(Angle45);
```

TRUNCATE(*realvalue*)

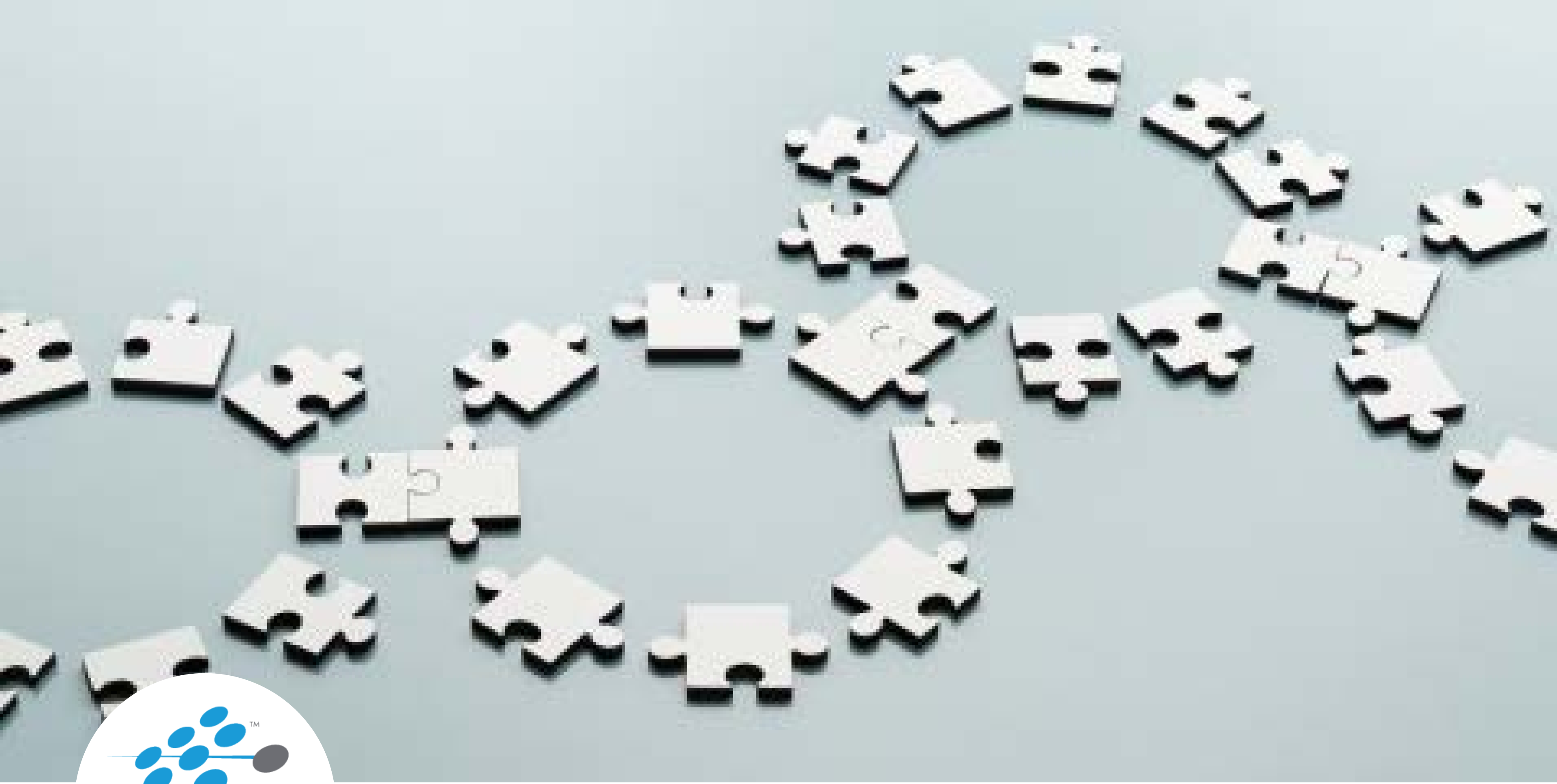
- ✓ *realvalue* – The floating point value to truncate.

```
Val := 3.14159;
ValTrunc := TRUNCATE(Val);  // Result is 3
```

Lesson Completed

Proceed to Lesson 24:

Value Definitions



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 24

Lab Exercise 9 – Value Definitions

Risk Solutions

ECL Definition Types - Value

Value –

A Value definition is an arithmetic or string expression with a single-valued result

```
ValueTrue        := 1;
FloridianCount   := COUNT(People(IsFloridian));
SeniorAvgAge     := AVE(People(IsSeniorCitizen), People.Age);
```

Lab Exercise 9:

Exercise 9

Value Definitions

- Reuse of BOOLEAN and SET definitions
- Use of COUNT
- Filtering
- Explicit Casting
- String Indexing

Lab Exercise 9:

Exercise Spec:

Create value definitions to determine the ratio of men born before 1980 and living in states beginning with 'M' to all men born before 1980 in the US. Do the same thing for females using the same criteria.

Steps:

1. Create a Builder Window Runnable file to display both values in a file named **BWR_PercentOlderGenderMStates**.
2. Use the EXPORTed **MeninMStatesPersons** RecordSet definition that you created in *Lab Exercise 8*.
3. You may find it handy to use the EXPORTed **SetMStates** set definition that you created in *Lab Exercise 7*.
3. Create two value definitions named **PercOlderMalesinMStates** and **PercOlderFemalesinMStates**. The values of each should be expressed as a percentage to two decimal places.
4. Percentages are essentially a ratio of counts multiplied by 100. Cast the results to a DECIMAL(5_2).
5. Extra credit for labeling your output results “Men_Percentage” and “Female_Percentage”.

Result Comparison:

Verify that the value definitions show the following results:

PercOlderMalesinMStates = 14.85,

PercOlderFemalesinMStates = 14.91.

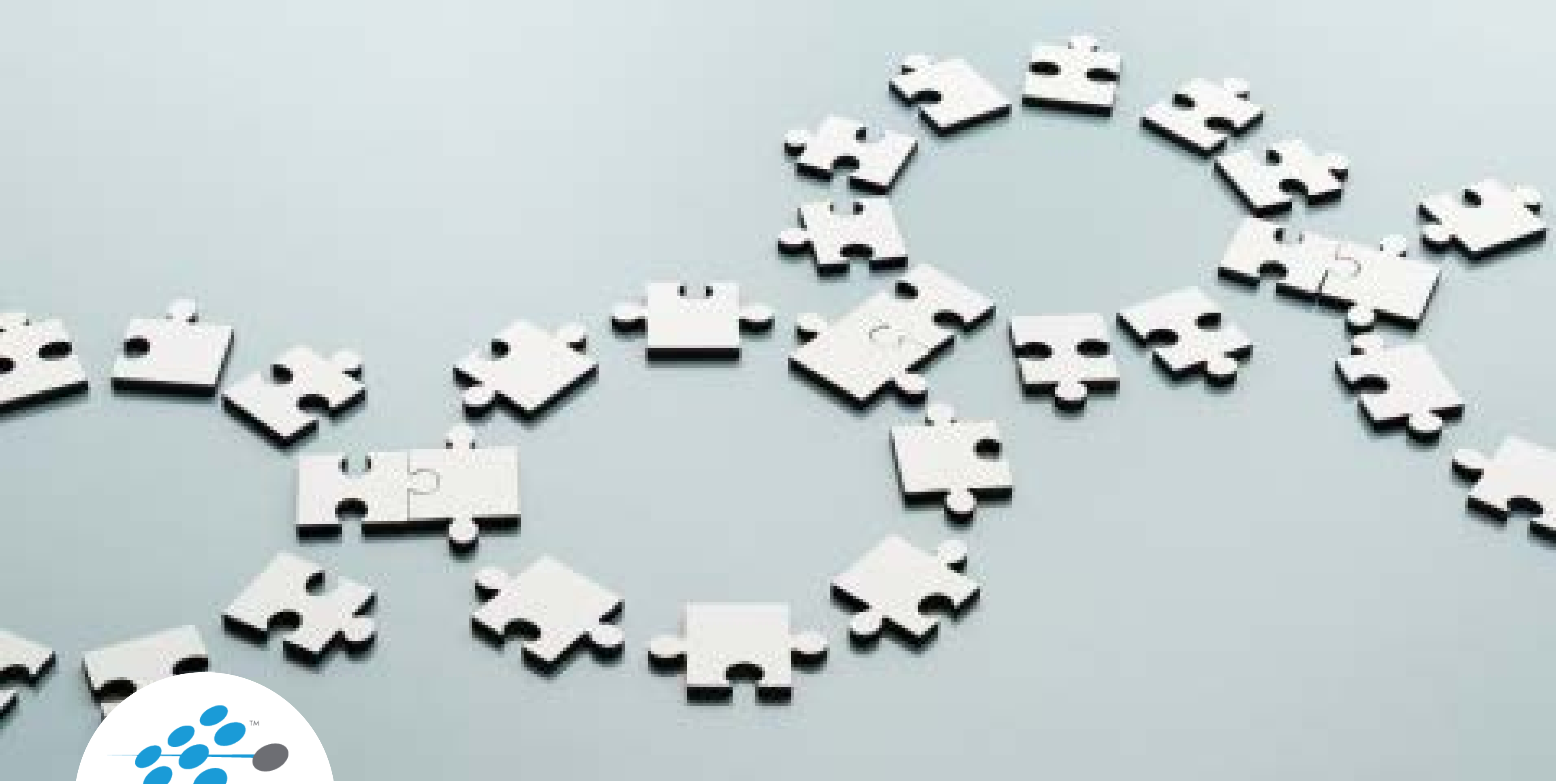
Lab Exercise 9 Solution:

```
IMPORT $;
//We want a ratio of men in "M" states born before 1980
//to men in ALL states born before 1980 (express in percentage to two decimal places)

OlderPersons := $.Persons.Birthdate[1..4] < '1980';

c1 := COUNT($.MeninMStatesPersons(OlderPersons));
c2 := COUNT($.Persons(Gender = 'M',OlderPersons));

PercOlderMalesinMStates := (DECIMAL5_2)(c1/c2 * 100);

//Females:

OlderFemaleinMStates := COUNT($.Persons(State IN $.SetMStates,OlderPersons,Gender = 'F')) -
                        COUNT($.MeninMStatesPersons(OlderPersons));

c3 := COUNT($.Persons(Gender = 'F',OlderPersons));
PercOlderFemalesinMStates := (DECIMAL5_2)(OlderFemaleinMStates/c3 * 100);

OUTPUT(PercOlderMalesinMStates,NAMED('Men_Percentage'));
OUTPUT(PercOlderFemalesinMStates,NAMED('Female_Percentage'));
```

Lesson Completed

Proceed to Lesson 25:

ECL FUNCTION Structure



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 25

Lab Exercise 10 – ECL Functions and the FUNCTION structure

LexisNexis®

Risk Solutions

Functions and Parameter Passing

Any ECL Definition can be defined to accept passed parameters (arguments), making it an **ECL Function,** which does not change the definition's basic type, just makes it more flexible

Defined parameters always appear in parenthesis following the ECL definition name

Multiple parameter definitions are separated by commas

DefinitionName([*ValueType*] *AliasName* [=*DefaultValue*]) := *expression*;

- ✓ The required *AliasName* names the parameter passed for use in the *expression*
- ✓ The optional *ValueType* defaults to INTEGER if omitted
- ✓ The optional *DefaultValue* allows you to omit the parameter

Functions and Parameter Passing – Examples:

```
Num25 := 25;
AddFive(INTEGER x=10) := x + 5;

NumResult1 := AddFive(Num25);   // Result is 30
NumResult2 := AddFive();        // Result is 15

//Passing a DATASET as a parameter
MyRec := {STRING1 Letter};
SomeFile := DATASET([{'A'},{'B'},{'C'},{'D'},{'E'}],MyRec);

FilteredDS(DATASET(MyRec) ds) := ds(Letter NOT IN ['A','C','E']);
           //passed dataset referenced as “ds” in expression
OUTPUT(FilteredDS(SomeFile));
```

Lab Exercise 10 (Part One and Two):

Exercise 10:

Standard ECL Functions (Passing Parameters)

Putting a lot of ECL together!

Part One:

- LENGTH
- TRIM

Part Two:

- String Indexing
- Aggregate Functions (COUNT, SUM, MIN,MAX,AVE)
- Inline DATASET

LENGTH

LENGTH(*expression*)

- ✓ *expression* – A string expression.

The **LENGTH** function returns the length of the string resulting from the expression.

```
INTEGER1 StrCnt := LENGTH('ABC' + 'XYZ');  // Result is 6
```

TRIM

TRIM(string*value* [, *flag*]**)**

- ✓ *stringvalue* – The string from which to remove spaces.
- ✓ *flag* – Optional. Specifies which spaces to remove. RIGHT removes trailing spaces (this is the default). LEFT removes leading spaces. RIGHT,LEFT removes both leading and trailing spaces. ALL removes all spaces.

The **TRIM** function returns the *stringvalue* with all trailing and/or leading spaces removed.

```
STRING20 StrVal := '   My ABC ';  // Contains 3 leading, 11 trailing spaces

VARSTRING StrVal1 := TRIM(StrVal, ALL); // Contains 3 characters
```

STRING Indexing

Strings (character sets) - may also be indexed to access individual characters within the string

```
MyString     := 'ABCDE';
MiddleChar := MyString[3];      //MiddleChar contains "C"
```

Substrings - may be extracted by using 2 periods to separate the beginning and ending element numbers to specify the substring to extract

```
MySubString1 := MyString[2..4];  // 'BCD'
MySubString2 := MyString[..4];    // 'ABCD'
MySubString3 := MyString[2..];    // 'BCDE'
```

Lab Exercise 10 (Part One):

Exercise Spec:

Create *two* EXPORT functions (i.e., ECL Definitions passing parameters). First, create a Value function to return a value limited to a maximum amount. The second function extracts the last four characters from any size string.

Steps:

1. Create an EXPORT value function called **fn_LimitValue(n, maxval)**
2. Use the IF function to determine if the passed n value is greater than the passed maxval. If true, return the maxval, otherwise return n.
3. Create an EXPORT value function called **fn_Right4(STRING s)**
4. USE the IF function to test the length of the passed STRING. If greater than 4, use string indexing with LENGTH and TRIM to extract the last 4 characters from the string. If 4 or less, simply return the passed string.

Result Comparison:

Open a Builder Window and test each function as follows:

```
IMPORT TrainingYourName;
TrainingYourName.fn_LimitValue(3,10);                //returns 3
TrainingYourName.fn_LimitValue(30,10);               //returns 10
TrainingYourName.fn_right4('here is a long string'); //returns 'ring'
TrainingYourName.fn_right4('33334-1502     ');       //returns '1502'
```

Lab Exercise 10 Solution (Part One):

fn_LimitValue.ECL

```
EXPORT fn_LimitValue(n,maxval) := IF(n > maxval, maxval, n);
```

fn_Right4.ECL

```
fn_Right4(STRING s) := IF(LENGTH(TRIM(s)) > 4,
                           s[LENGTH(TRIM(s))-3..],
                           s);

fn_Right4('thisismystring');     //returns ring
fn_Right4('33334-1502    ');     //returns 1502
```

Lab Exercise 10 (Part Two):

Aggregate Function – Quick Review

- **COUNT(***recordset* **)**
- **SUM(***recordset* , *value* **)**
- **AVE(***recordset* , *value* **)**
- **MIN(***recordset* , *value* **)**
- **MAX(***recordset* , *value* **)**

INLINE DATASET

name := **DATASET(***recordset*, *recorddef***)**

- ✓ *name* – The definition name by which the file is subsequently referenced.
- ✓ *recordset* – A set of data records contained within square brackets (indicating a set definition). Within the square brackets, each record is delimited by curly braces ({}) and separated by commas. The fields within each record are comma delimited.
- ✓ *recorddef* – The RECORD structure of the dataset.

```
NamesRec := RECORD
    STRING20 first_name;
    STRING20 last_name;
END;
Names := DATASET([{'John','Jones'}, {'Jane','Smith'}], NamesRec);
```

INLINE DATASET: Example

```
IMPORT $;
c1 := COUNT($.Persons(DependentCount=0));

c2 := COUNT($.Persons);

c3 := (INTEGER)(((c2-c1)/c2)*100.0);

d := DATASET([{'Total Records',c2},
        {'Recs=0',c1},
        {'Population Pct',c3}],
    {STRING15 valuetype,INTEGER val});

OUTPUT(d);
```

Special Structures – FUNCTION

FUNCTION Structure

```
[resulttype]  funcname( parameterlist ) := FUNCTION
    code;
    RETURN returnvalue;
END;
```

- ✓ *resulttype* – The return value type of the function. If omitted, the type is implicit from the *returnvalue* expression.
- ✓ *funcname* – The ECL definition name of the function.
- ✓ *parameterlist* – The parameters to pass to the *code* – available to all definitions in the FUNCTION's *code*.
- ✓ *code* – The definitions in the FUNCTION's process.
- ✓ **RETURN** – Specifies the function's *returnvalue* expression.
- ✓ *returnvalue* – The value, expression, recordset, row (record), or action to return.

Special Structures – Without FUNCTION

```
isTradeRateEQSince(STRING1 rate,age) := Trades.trd_rate = rate AND
                                        ValidDate(trades.trd_drpt) AND
                                        AgeOf(trades.trd_drpt) <= age;

isPHRRateEQSince(STRING1 rate,age) := EXISTS( Prev_rate(phr_rate = rate,
                                              ValidDate(phr_date),
                                              AgeOf(phr_date) <= age,
                                              phr_grid_flag = TRUE));

isAnyRateEQSince(STRING1 rate,age) := isTradeRateEQSince(rate,age)
                                      OR
                                      isPHRRateEQSince(rate,age);
```

Special Structures – with FUNCTION

```
EXPORT isAnyRateEQSince(STRING1 rate,age) := FUNCTION

isTradeRateEQSince := Trades.trd_rate = rate AND
                      ValidDate(trades.trd_drpt) AND
                      AgeOf(trades.trd_drpt) <= age;

isPHRRateEQSince := EXISTS(Prev_rate(phr_rate = rate,
                                     ValidDate(phr_date),
                                     AgeOf(phr_date) <= age,
                                     phr_grid_flag = TRUE));

  RETURN isTradeRateEQSince OR isPHRRateEQSince;
END;
```

Lab Exercise 10 (Part Two):

Exercise Spec:

Create a new ECL Function that returns Count, Sum, Average, Minimum and Maximum aggregate values for the **Persons** *DependentCount* field.

Steps:

1. Create an EXPORT ECL Definition that will use a FUNCTION structure. Pass the Persons DATASET and the field to aggregate as follows:

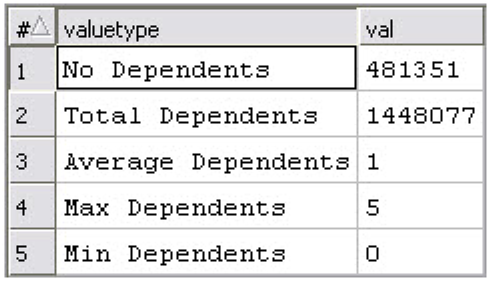
```
EXPORT fn_Aggregates(DATASET(RECORDOF($.Persons)) DS,INTEGER FieldName) := FUNCTION
```

2. USE COUNT, SUM, AVE, MIN and MAX as the field aggregates. The COUNT should return all persons who have no dependents.

3.Use an INLINE DATASET definition to produce the output of the function.

4. The INLINE Dataset should have a least two fields, one holding the aggregate results, and the other labeling the results.

For example, the expected output should look like this:



#	valuetype	val
1	No Dependents	481351
2	Total Dependents	1448077
3	Average Dependents	1
4	Max Dependents	5
5	Min Dependents	0

Result Comparison:

Open a new builder window, and test your function as follows:

```
IMPORT TrainingYourName AS X;
X.fn_Aggregates(X.Persons,X.Persons.DependentCount);
```

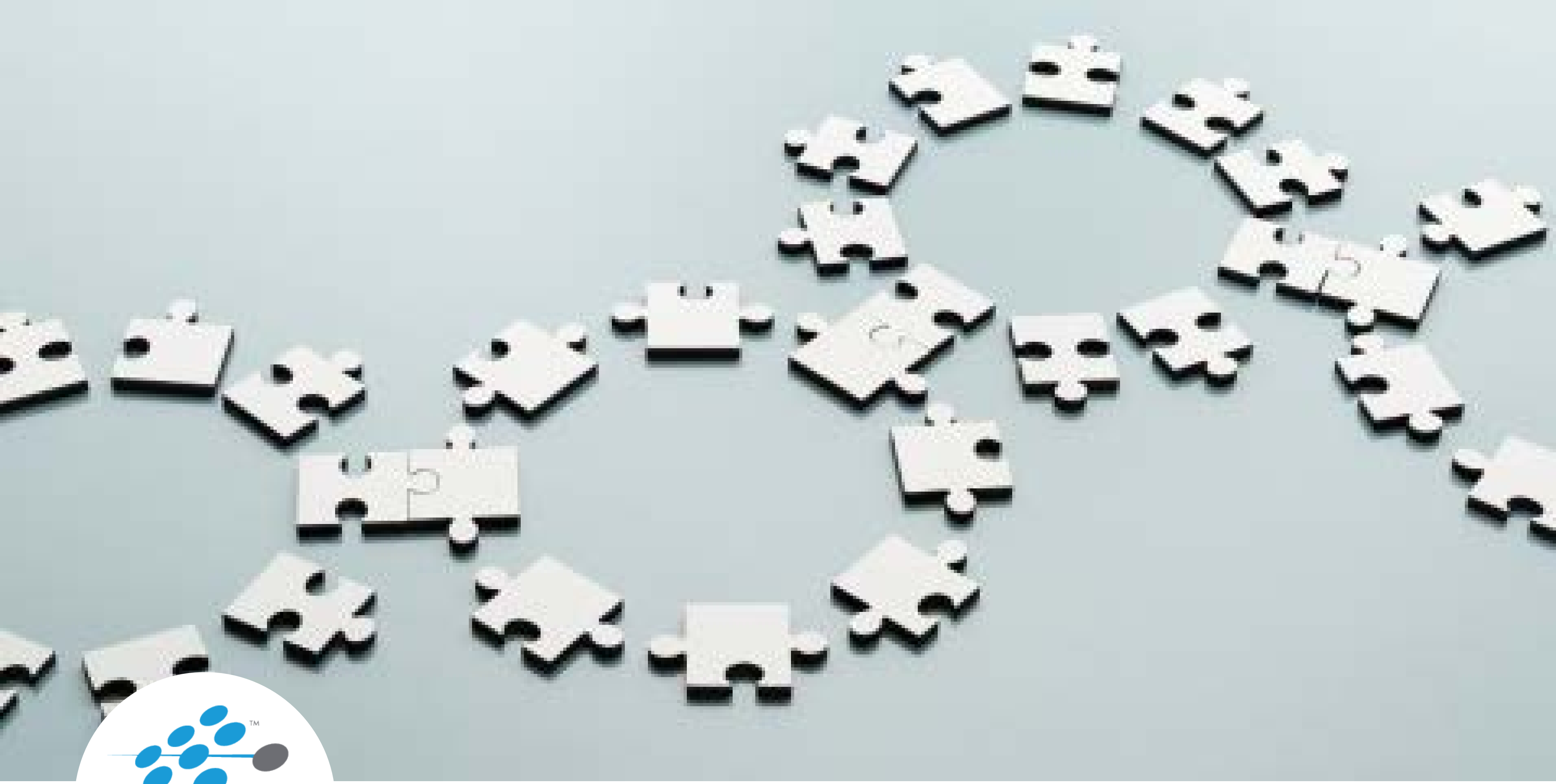
Lab Exercise 10 Solution (Part Two):

```
IMPORT $;
EXPORT fn_Aggregates(DATASET(RECORDOF($.Persons)) DS,INTEGER FieldName) := FUNCTION

 c1 := COUNT(DS(Fieldname = 0));
 s1 := SUM(DS,Fieldname);
 a1 := AVE(DS,Fieldname);
 m1 := MAX(DS,Fieldname);
 m2 := MIN(DS,Fieldname);

 d := DATASET([{'No Dependents',c1},
              {'Total Dependents',s1},
              {'Average Dependents',a1},
              {'Max Dependents',m1},
              {'Min Dependents',m2}],
              {STRING20 valuetype,INTEGER val});
 RETURN(d);
END;
```

Lesson Completed

Proceed to Lesson 26:

The MODULE Structure



HPCC SYSTEMS®

HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 26

The MODULE Structure

LexisNexis®

Risk Solutions

Special Structures – MODULE

MODULE Structure

modulename **[** (*parameterlist*) **]** := MODULE

definitions;

END;

- ✓ *modulename* – The ECL definition name of the module.
- ✓ *parameterlist* – The parameters available to all *definitions*.
- ✓ *definitions* – The ECL definitions that comprise the module. These definitions may receive parameters, and may include actions (such as OUTPUT).

ECL Definition Visibility (Scoping)

Global (sort of) –

```
EXPORT PeopleCount := COUNT(People);   //Available "globally"
```

Module –

```
SHARED StateCount := 50;   //Available only within its own module
```

Local –

```
Num5 := 5;         // Local Scope – Available only to NumTotal
EXPORT NumTotal := Num5 + 10 + StateCount;
```

Reserved Keywords

IMPORT folderlist

- ✓ folderlist – A comma-delimited list of folder names. This may contain the keyword AS followed by an alias to rename a specific folder in the list.

```
IMPORT $; //Alias current folder name with $
IMPORT * FROM TrainingYourName; //override qualification
IMPORT Companies, STD;
IMPORT  YellowPages AS YP;
```

Reserved Keywords (cond)

EXPORT *definition*

- ✓ *definition* – A definition file in a repository folder.

```
EXPORT File_Company := DATASET('Company',
                               Layout_Common,FLAT);
```

Reserved Keywords (cond)

SHARED *definition*

- ✓ *definition* – An ECL definition within a file.

```
SHARED STRING8 Version := '20021101';
```

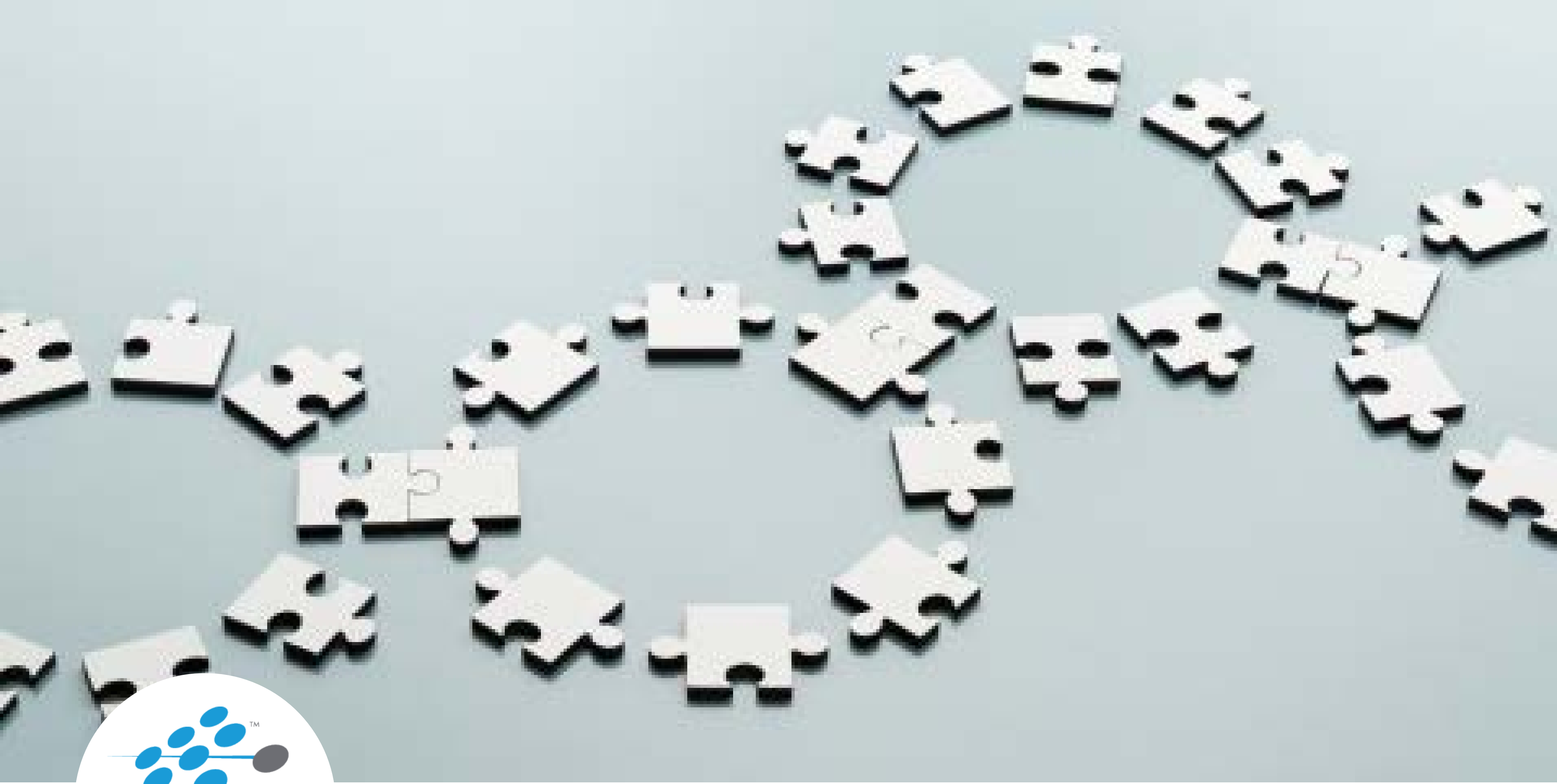
Special Structures – MODULE Example

```
EXPORT filterDataset(STRING search,
                     BOOLEAN onlyOldies) := MODULE
   f := namesTable;
   SHARED g := IF (onlyOldies, f(age >= 65), f);
   EXPORT included := g(surname = search);
   EXPORT excluded := g(surname <> search);
END;
filtered := filterDataset('Halliday', true);
OUTPUT(filtered.included,,NAMED('Included'));
OUTPUT(filtered.excluded,,NAMED('Excluded'));
```

Lesson Completed

Proceed to Lesson 27:

Duplicate Record Processing



HPCC Systems: Introduction to the Enterprise Control Language

Lesson 27

Lab Exercise 11 - Duplicate Record Processing

Risk Solutions

SORTing Your Record Sets

SORT(*recordset, value* **)**

recordset – The set of records to process.

value – An expression or key field in the *recordset* on which to sort. A leading minus sign (-) indicates a descending order sort. You may have multiple *value* parameters to indicate sorts within sorts.

```
Business_Contacts_Sort := SORT(Business_Contacts_Dist,
                               company_name, company_title,
                               lname, fname, mname, name_suffix,
                               zip, prim_name, prim_range);
HighestBals := SORT(ValidBalTrades, -trades.trd_bal);
UCC_Sort    := SORT(UCC_Dist, file_state, orig_filing_num);
```

More Recordset Functions – SORT Example

Training_Examples.SORT_Example

More Recordset Functions - DEDUP

DEDUP(*recset* [,*condition* [,**ALL**][,**KEEP** *n*] [,*keep*]]**)**

recset – The set of records to process.

condition – The expression that defines "duplicate" records.

ALL –Matches all records to each other using the *condition*, not just adjacent records.

KEEP *n* –Specifies keeping *n* number of duplicates. The default is 1.

keep –LEFT (the default) keeps the first and RIGHT keeps the last.

```
Company_Dedup := DEDUP(Company_Init(zip<>0), company_name,
                       zip, prim_name, prim_range, sec_range, ALL);
```

LEFT and RIGHT Keywords

LEFT.*field* / RIGHT.*field*

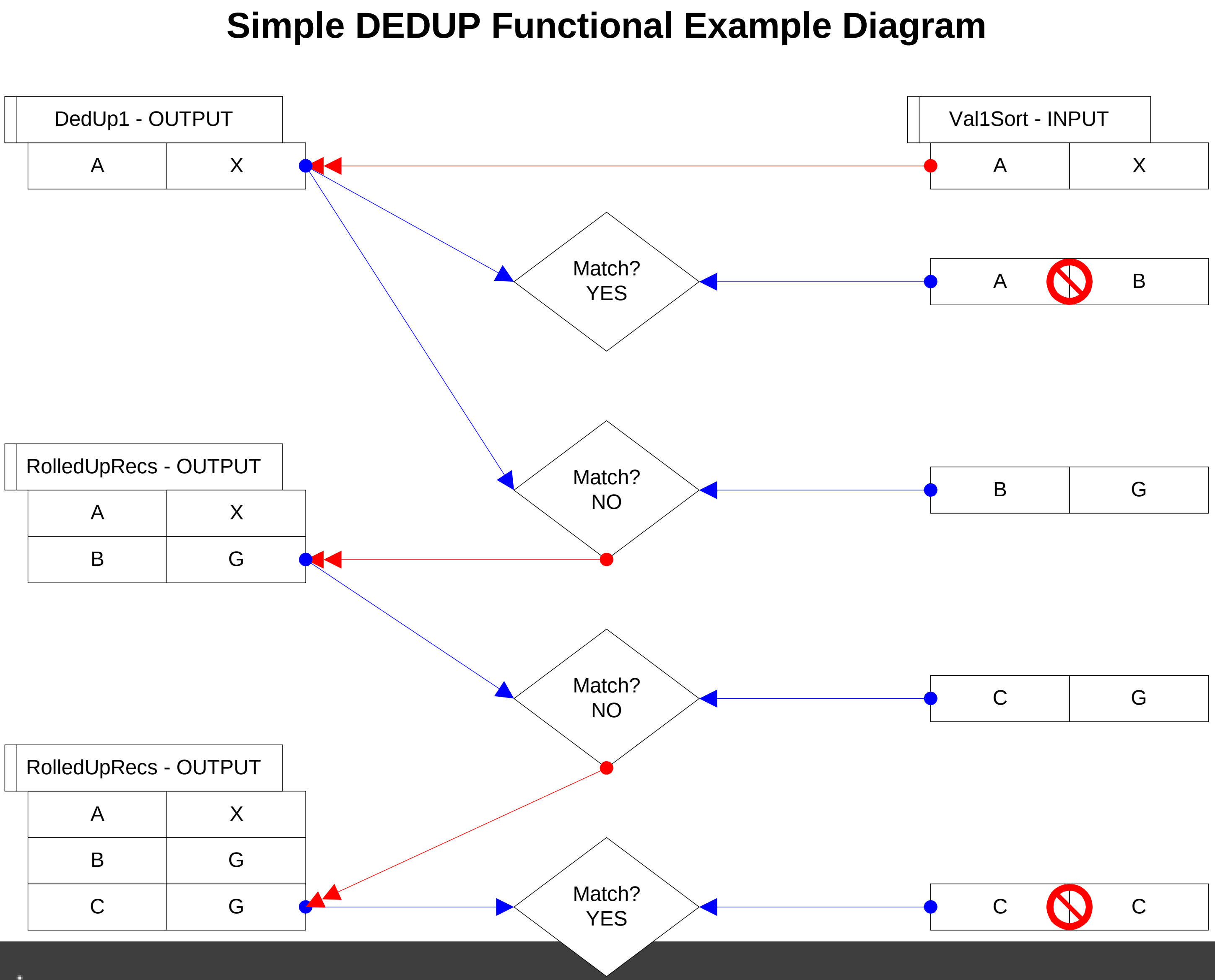
The **LEFT** and **RIGHT** keywords qualify which record a *field* is from in those operations that use pairs of records, such as DEDUP, JOIN, etc. This helps prevent any code ambiguity for the compiler.

```
SortedRecs := SORT(Persons,last_name,first_name);
DeDupedRecs := DEDUP(SortedRecs,
                     LEFT.last_name = RIGHT.last_name AND
                     LEFT.first_name = RIGHT.first_name);
```

More Recordset Functions – DEDUP Example

Training_Examples.DEDUP_Example

DEDUP Functional Diagram:



GROUP Function (used with DEDUP ALL)

GROUP(*recordset* [, *breakcriteria* [, **ALL**]]**)**

recordset – The set of records to fragment.

breakcriteria –An expression that specifies how to fragment the *recordset*. If omitted, the *recordset* is ungrouped.

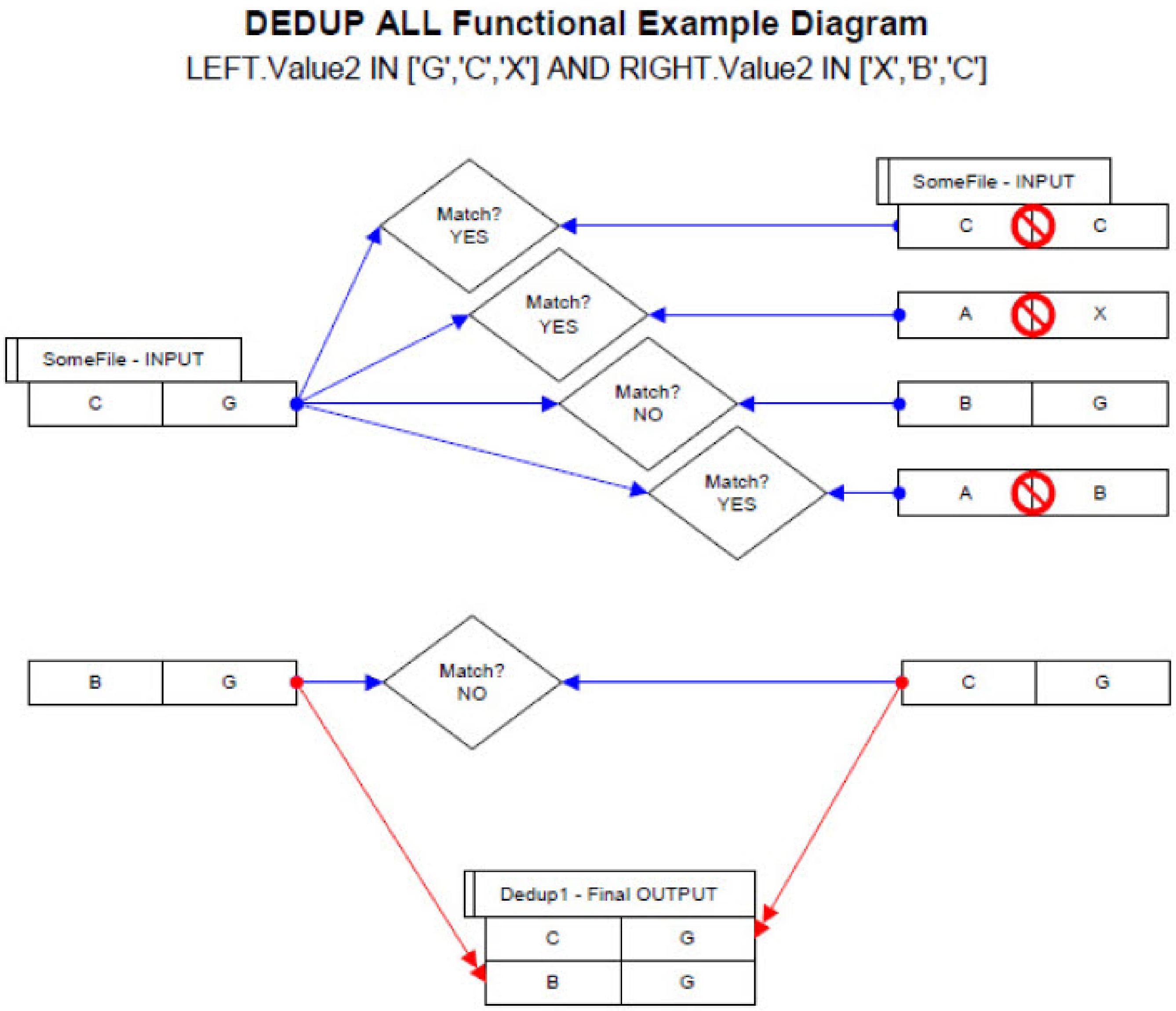
ALL –Indicates the *breakcriteria* is applied without regard to any previous order. If omitted, GROUP assumes the *recordset* is already sorted in *breakcriteria* order.

```
Grpd      := GROUP(UCCSort, file_state, orig_filing_num, LOCAL);
Dedup_1 := DEDUP(Grpd, (LEFT.filing_date <> RIGHT.filing_date
                                   OR  LEFT.filing_type <> RIGHT.filing_type),ALL);
```

More Recordset Functions – DEDUP with ALL

Training_Examples.DEDUP_ALL_Example

DEDUP with ALL Functional Diagram:



Workflow Services – PERSIST (Saving progress)

PERSIST

definition := *expression* : **PERSIST**(*file*);

definition – The name of an ECL definition.

expression – The ECL definition

file – A string constant defining the logical filename in which the results of the definition's *expression* are stored.

Lab Exercise 11:

Exercise 11:

- Duplicate Record Processing
- Use of SORT and DEDUP

Lab Exercise 11:

Exercise Spec:

Remove duplicate records from the Persons DATASET, based on Last and First Names. Calculate the number of unique records after the duplicate records have been removed.

Steps:

1. Create an EXPORT RecordSet definition called **DedupPersons**.
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. Use DEDUP to remove all duplicate *LastName*, *FirstName* records in the Persons dataset. Always keep the record with the lowest *RecID*.
4. Use the PERSIST workflow service in the **DedupPersons** definition so the results will not have to be recalculated on subsequent usage.
5. The PERSIST name must start with **~CLASS**, followed by your initials, followed by **PERSIST::DedupPersons** as in this example:

```
~CLASS::XX::PERSIST::DedupPersons
```

Result Comparison:

Open an ECL Builder window and submit the **DedupPersons** definition. In addition, use COUNT in the Builder window to verify that the DEDUP was performed correctly and that there are now **841327** unique names in the Persons DATASET.

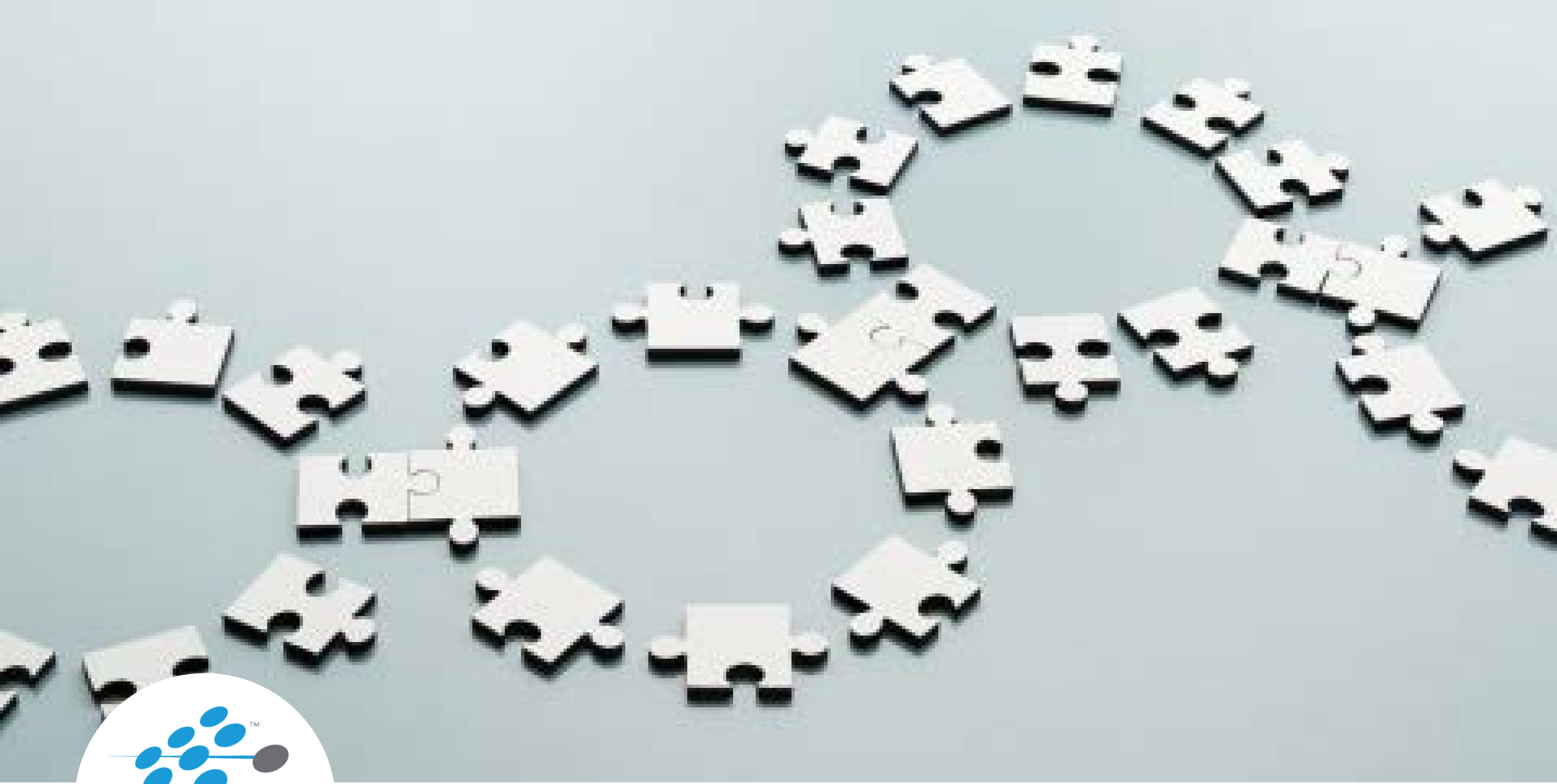
Lab Exercise 11 Solution:

```
IMPORT $;
SortAccounts := SORT($.Persons,LastName,FirstName,RecId);
EXPORT DedupPersons := DEDUP(SortAccounts,
                        LEFT.Lastname = RIGHT.LastName AND
                        LEFT.Firstname = RIGHT.FirstName)
      :PERSIST('~CLASS::BMF::TEMP::DedupPersons');
```

Lesson Completed

Proceed to Lesson 28:

Output to Cluster and the Landing Zone



HPCC Systems:

Introduction to the Enterprise Control Language

Lesson 28

Lab Exercise 12 & 13 - Output to the Cluster and the Landing Zone

Risk Solutions

Basic Actions

OUTPUT

[*name* :=] **OUTPUT(***recordset* [,*format*] [,*file* [,OVERWRITE]]**)**

name – Optional definition name for this action

recordset – The set of records to process

format – The format of the output records: a previously defined RECORD structure, or an "on-the-fly" record layout enclosed in { } braces.

file – Optional name of file to write the records to. If omitted, formatted data stream returns to the command line or Query Builder program.

OVERWRITE – Allows file to be overwritten if it exists

OUTPUT Examples:

```
OUTPUT(Accounts); //Equivalent to: Accounts;

OUTPUT(Persons,{FirstName,LastName}, NAMED('Names_Only'));

//***************************************

OUTPUT(BHF_Out,,'OUT::Business_Header_' +
            Business_Header.Version, OVERWRITE);

OUTPUT($.DedupPersons,, '~ONLINE::XXX::DedupPersons', OVERWRITE);

OUTPUT($.DedupPersons(State = 'FL'),,
                 '~ONLINE::XXX::DedupFloridaPersons', OVERWRITE);
```

Lab Exercise 12:

Exercise 12:

Writing a new recordset to disk

- Use of OUTPUT

Lab Exercise 12:

Exercise Spec:

Take the "deduped" Persons recordset definition that you created in Exercise 11 (**DedupPersons**) and write it to a new file on disk that you will define. We will use this file to "despray" the information back to the landing zone in Exercise 13.

Steps:

1. Create a new Builder window definition named **BWR_OutDedupPersons** (and comment out or delete the export line, just as you did in previous exercises).
2. IMPORT all definitions from your Training Module to eliminate the need to fully qualify your definitions.
3. The OUTPUT filename must start with *~ONLINE*, followed by your initials, followed by *OUT::DedupPersons* as in this example:

```
~ONLINE::XX::OUT::DedupPersons
```

4. Use the **OVERWRITE** option in your OUTPUT statement to allow rewrites of this file when necessary.

Best Practices Hint:

The use of BWR in the definition name implies "Builder Window Runnable" code that is designed to run in a Builder window, but is stored in the Repository in the same manner as other ECL definitions. This implies several things:

It must contain at least one action.

All referenced definitions from the Repository must be fully qualified.

The BWR code contains no EXPORT or SHARED definitions.

Result Comparison:

Use a Builder window to execute the query, and then look in the ECL Watch Logical Files list to find the newly generated file and ensure that its size is about 127,040,377 and that there are 841,327 records.

Lab Exercise 12 Solution:

```
IMPORT $;

 OUTPUT($.DedupPersons,,
        '~ONLINE::XXX::OUT::DedupPersons',OVERWRITE);
        //replace XXX with your initials
```

Lab Exercise 13:

Exercise 13:

Despraying – Instructor will demonstrate

- ECL Watch
- Browse Logical Files
- Context Menu Button (Grey Button)
- Despray Options

Lesson Completed

Course Completed!

Next Course:

Introduction to THOR

The Extract, Transform and Load Process