# SPL Examples for Beginners

This package (SPL-Examples-For-Beginners.tar.gz) includes 122 different examples. These are simple to moderately complex examples that will help developers who are just beginning to wrap their minds around IBM Streams Processing Language (SPL). Along with a great number of Streams SPL programming documentation files and advanced examples shipped with the Streams product, beginners can use this package of examples as an additional learning aid.

All the examples listed below are self contained and every example focuses on a particular aspect of the SPL feature set. These individual project directories can be imported directly into Eclipse-based Streams Studio development tool and executed from there. A procedure to import them into Streams Studio is explained at the end of this file.

The goal here is to make the beginners get their feet wet via these simple SPL examples that produce verifiable results.

[These examples were developed and tested using the Streams Studio IDE running on 64 bit Red Hat Enterprise Linux 6.3 and higher.]

The following is a catalog of all the examples with a short description about what is covered in each one of them.

Use the InfoSphere Streams perspective available inside the Streams studio to work with the following projects.

### 001_hello_world_in_spl

This example is the simplest possible SPL application. It uses a Beacon operator to generate tuples that carry "Hello World" messages. A custom sink operator receives the tuples from Beacon and displays it on the console.

### 002_source_sink_at_work

This example shows how a FileSource operator can be used to read CSV formatted records from a file and then receive those tuples in a FileSink to be written to a file in the data directory of this application.

### 003_sink_at_work

This example shows how FileSink and Custom sinks can be employed in applications. It also shows how a Beacon operator can be used to customize tuple attributes. In addition, it introduces the Filter operator to route the incoming tuples by inspecting their attributes using a conditional statement specified in the filter parameter.

### 004_delay_at_work

This example shows how a Delay standard toolkit operator can be used to delay a stream. This example also introduces the Custom operator that can be used to perform custom logic. You can also notice the use of a state variable that is mutable inside the Custom operator. It also shows how to create a new tuple on the fly and do your own submissions onto the output ports.

### 005_throttle_at_work

This example shows how a stream can be throttled to flow at a specified rate. This example also mixes other operators such as Beacon, Custom, and FileSink.

### 006_barrier_at_work

This example shows how to synchronize the incoming tuples using a Barrier operator. It uses a bank deposit/debit scenario to split the deposit/debit requests, perform that account activity, and then combine the post-activity result with the incoming requests. Barrier operator does what is needed to accomplish that i.e. it waits for the streams to arrive at all the configured input ports before emitting an output tuple.

### 007_split_at_work

This example shows how a Split operator can be used to split the incoming tuples based on a key. In this example, the split condition (which tuples comes out on which port) is pre configured through a text file. Alternatively, one can compute the index of the output port on the fly inside the Split operator parameter section.

### 008_get_submission_time_value

This example shows how the tuple attributes can be assigned values that were supplied by the user at the application/job submission time. It employs the getSubmissionTimeValue function to obtain different values made of different SPL data types.

### 009_custom_operator_using_get_submission_time_value

This example demonstrates how to assign tuple attributes at the time of job submission inside a custom operator. When the incoming tuples arrive at the Custom operator in this example, values entered by the user at the application startup are assigned to the tuple attributes.

### 010_get_compiler_time_value

This example shows how arguments supplied during the application compile time can be accessed inside of the SPL applications. In Streams Studio, you can enter these values by editing the active build configuration. Additional SPL compiler options can be added in the resulting build configuration dialog.

### 011_compiler_intrinsic_functions

Streams compiler provides several intrinsic functions to query the SPL filename, file path, absolute path of the directory, source code line number, composite instance name etc. This example shows the use of the compiler intrinsic functions inside of a Functor operator.

### 012_filter_functor_at_work

This example puts the two commonly used standard toolkit operators to work. They are Filter and Functor. Filter allows you to route tuples based on conditional checks. It provides two output ports to send the matched tuples on the first output port and the unmatched tuples on the second output port. Functor operator allows us to transform the incoming tuple attributes and then to send it on many different output ports with different stream schemas.

### 013_punctor_at_work

This example shows how a Punctor operator could be used in an application. Punctor operator allows us to transform the input tuples and then inject puncuation markers either before or after the output tuple as configured.

### 014_sort_at_work

This example shows the use of the Sort operator in the context of an application. Sort operator is highly configurable with all kinds of windowing support. In this example, the following window configurations are applied for sorting the incoming tuples.

a) Count-based tumbling window.
b) Time-based tumbling window.
c) Punctuation-based tumbling window.
d) Delta-based tumbling window.
e) Count-based sliding window.

### 015_join_at_work

This example shows one of the power-packed standard toolkit operators; i.e. Join. This operator is so versatile that it is hard to do justice in explaining it thoroughly in a simple example such as this one. This example provides coverage to the following Join operator features.

a) Inner Join
b) Inner (Equi) Join
c) Left Outer Join
d) Right Outer Join
e) Full Outer Join

### 016_aggregate_at_work

This example shows off yet another powerful standard toolkit operator named the Aggregate. It is very good in computing on the fly aggregate values after collecting a set of tuples. Tuples are grouped based on tumbling and sliding windows with partitioned variants. This example also shows how to use the built-in assignment functions provided by this operator to compute regular statistical calculations such as min, max, average, standard deviation etc.

### 017_filesource_filesink_at_work

We have used the FileSource and the FileSink operators in other examples before. However, this example shows off the following intriguing features that will become handy in a lot of practical situations.

a) Automatic deletion of a file after the FileSource finishes reading all the records.
b) Flushing the sink file on demand after writing a certain number of tuples.
c) Ability of the FileSource to move the file once it reads all the content in that file.
d) Creating a fresh and new output sink file after writing a certain number of tuples.
e) Ability of the FileSource to keep reading from a hot file as new CSV records get written to the end of that file.

### 018_directory_scan_at_work

This example demonstrates one of the important features desired in the real world (mostly in the Retail banking and in the Telco industries). In many real-world scenarios, they still work via files and such files get dropped into a directory for processing. It is shown here how the DirectoryScan operator picks up a new file as soon as it appears inside an input directory. (Apply caution if huge files are copied to the watch directory. DirectoryScan may detect that big file copy as multiple new files and output multiple tuples with the same file name.)

### 019_import_export_at_work

This example demonstrates how two different SPL applications can share streams between them. This is an important feature that is elegantly done using two pseudo operators called Export and Import. This application also shows how two different main composites can be part of the same application by using two different namespaces. As an aside, there is also a demonstration of using a Custom operator to customize the Beacon generated tuples by involving state variables.

### 020_metrics_sink_at_work

This example shows how one can use the MetricsSink standard toolkit operator to create application-specific custom metrics that can be viewed in real-time when the application is running. Viewing of custom metrics is typically done inside Streams Explorer view of the Streams Studio or by using the capturestate option in streamtool.

### 021_pair_at_work

This example shows off the Pair operator that is used for pairing tuples arriving on different input ports. Only when all the tuples arrive at all the input ports, this operator will emit them one after the other in their order of arrival.

### 022_deduplicate_at_work

This example describes the use of an important operator that is highly applicable in many Telco scenarios. That operator is called DeDuplicate, which eliminates duplicate tuples for a specified duration of time. It also has an optional second output port on which duplicate tuples could be sent out for additional processing.

### 023_union_at_work

This example demonstrates an utility operator called Union. This operator combines all the tuples from several input ports as they arrive and emits a single output stream. All the input ports must have a schema that contains attributes of the same name and type as those of the output port. The order of the attributes in the input ports need not match the order in the output port.

### 024_threaded_split_at_work

This example demonstrates an important standard toolkit operator named ThreadedSplit. It is a multi-threaded split that is different from the other content-based Split operator. ThreadedSplit uses its own algorithm to split the incoming tuples to the available output ports to improve concurrency. This will speed up the distribution of tuples by using individual threads assigned to each of the output ports.

### 025_dynamic_filter_at_work

This example deals with an interesting standard toolkit operator called DynamicFilter. This operator is a special version of the Filter operator that you have already seen in another example; it decides at runtime which input tuples will be passed through, based on the control input it receives. This operator is applicable in many real-life scenarios. This example also demonstrates using a second composite operator to perform a sub-task that the main composite will make use of. There is also coverage to show how the second composite can take its own operator parameters.

### 026_gate_at_work

This is an example that uses the Gate operator from the standard toolkit. This operator delays the incoming tuples until a downstream operator signals with an acknowledgment to receive any further tuples. This is a great way to have a feedback through which we can control the rate at which tuples are passed through. (Please refer to another example named 905_gate_load_balancer that shows the effectiveness of the Gate operator in combination with the ThreadedSplit operator to provide load balancing the incoming tuples.)

### 027_java_op_at_work

This example shows an important operator that brings Java into the C++ dominated world of Streams!!! That operator is called JavaOp, which is used to call out to other operators implemented in Java using the Java Operator API. In this example, we will have a tiny Java logic that will calculate the current time and add that time string to a tuple attribute and output that tuple. There is another example that shows the Java primitive operator that is different from the JavaOp operator.

### 028_multiple_composites_at_work

This example shows the use of multiple composites in a single application. There is a main composite that in turn uses two other composites. This application shows how the additional composites in different namespaces get included into the main composite via the "use" directive. It also demonstrates how the additional composites can accept their own operator parameters. It teaches the basics of an important feature that will come handy when big applications need to be componentized.

### 029_spl_functions_at_work

This example shows how helper and utility functions can be written using the SPL language. It also shows how such SPL functions can be put to use inside the context of an application. Learning this simple concept will go a long way in doing a lot of neat stuff in real-world applications.

### 030_spl_config_at_work

This example introduces one of the must-learn features of the SPL language. SPL language offers an extensive list of options to do configuration at the operator level as well as at the composite level. This application attempts to sprinkle many of the available configuration parameters as shown below.

a) host
b) hostColocation
c) partitionColocation
d) placement
e) threadedPort and queue
f) relocatable and many more.

In addition, this example shows how to make this application toolkit dependent on another (025_dynamic_filter_at_work) SPL toolkit project.

### 031_spl_mixed_mode_at_work

This example shows a cool SPL feature called mixed-mode support. In this, developers can mix PERL code islands inside of an SPL application. Mixed-mode enables the easy parameterization of SPL applications. This example gives a slight flavor of how a PERL code snippet inter-mixed with SPL allows us to parameterize the SPL Stream names and the number of output stream definitions for an SPL operator.

### 032_native_function_at_work

This application shows how native functions written in C++ can be called within an SPL application.

There are two ways in which native functions can be written in C++.

1) Code for the C++ functions can be written in a C++ header file.

2) C++ functions can be written outside of the SPL project and packaged into a shared library (.so) file. All the SPL developer will have to work with are an .so file and a C++ header file.

This application demonstrates incorporating native functions built in both of those ways.

*[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED NativeFunctionLib THAT IS DESCRIBED BELOW.]*

### 033_java_primitive_operator_at_work

This example shows how a Java primitive operator is created from scratch. Java primitive operator is different from JavaOp that you have seen earlier in a different example. Java primitive operator is a first

class operator in SPL, whereas JavaOp only permits a callout to another Java operator. In addition, Java primitive operator has the advantage of keeping its name as the operator's runtime instance name.

*[THIS EXAMPLE HAS A COMPANION JAVA PROJECT NAMED RSS_Reader_Primitive THAT IS DESCRIBED BELOW.]*

### 034_odbc_adapters_for_db2_at_work

This example shows the use of the three Streams ODBC adapters. Those operators are ODBCSource, ODBCAppend, and ODBCEnrich. The code in this example is written to access a particular test DB2 database inside IBM. You have to create your own DB2 database and tables to make this application work in your environment. After creating your own database and tables, you have to change the etc/connections.xml file in this application's directory to match your database/table names, userid, and password. You also have to make changes in the SPL code using your database information for all the three ODBC operator invocations.

### 035_c++_primitive_operator_at_work

This example shows the steps required to create a C++ primitive operator from scratch. In this application, a C++ primitive operator model XML file can be explored to learn how the different fields in that file are configured. Then, the code generation template header and implementation files (*_h.cgt and *_cpp.cgt) can be browsed to learn about the primitive operator logic. Additionally, this example demonstrates about including a Java operator and a C++ primitive operator as part of the application flow.

### 036_shared_lib_primitive_operator_at_work

This example demonstrates two important techniques that will be commonly used in real-world use cases.

1) Creating a C++ primitive operator.

2) Calling a function available inside a .so shared library from the C++ primitive operator logic.

Application logic here is to receive input tuples as hostnames and then make the C++ primitive operator logic invoke a shared library function that does a name server lookup.

*[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED PrimitiveOperatorLib THAT IS DESCRIBED BELOW.]*

### 037_odbc_adapters_for_solid_db_at_work

This example shows the use of the three Streams ODBC adapters for connecting to a SolidDB in-memory database. Those operators are ODBCSource, ODBCAppend, and ODBCEnrich. The code in this example is written to access a particular test SolidDB database inside IBM. You have to create your own SolidDB database and tables to make this application work in your environment.

After creating your own database and tables, you have to change the ./etc/connections.xml file in this application's directory to match your database/table names, userid, and password. You also have to make changes in the SPL code using your database information for all the three ODBC operator invocations.

### 038_spl_built_in_functions_at_work

This is a very simple example that showcases a random collection of powerful built-in SPL functions that are available out of the box. This application demonstrates how time, math, and collection type functions can be used inside of an SPL application.

### 039_application_set_at_work

This example shows how multiple SPL applications can be grouped together so that they can be started, monitored, and stopped together. There is no code that needs to be written to accomplish this grouping.

One can simply create an SPL application set project inside the Streams Studio and then right-click on this project to get a dialog box displaying all possible applications in your workspace eligible to be grouped. You can now select the SPL projects that need to be grouped together. It is important to note that only those applications with Distributed active build configuration are eligible for the application set grouping.

In this example, you will notice that the 019_import_export_at_work and the 030_spl_config_at_work projects are added to the application set. You can now right click on the 039_application_set project and build/start/stop all of its member applications together.

### 040_ingest_data_generation_in_spl

This example shows how SPL provides rich features to generate synthetic data required for large scale testing. Many real-life applications in the Telco and the Retail Banking sectors consume large amounts of daily business data through CSV formatted text files. There could be huge amounts of CDR data from several telecom circles or daily transaction data for millions of accounts in a retail bank.

While building and testing the SPL applications, it will become necessary to generate such ingest data files with artificial data that is close enough to be realistic. This application shows how such large amounts of data in several thousands of files can be created very quickly using the SPL standard toolkit operators as well as the SPL file IO and math random built-in functions.

### 041_real_time_streams_merger

This example shows how two or more incoming streams with a common schema can be merged to flow in a sequence one after the other. This merger is done using a common tuple attribute in those multiple incoming streams as a key. We will use a C++ primitive operator called OrderedMerger that is included in this project. In order for the OrderedMerger to work correctly, it is assumed that multiple input streams for this primitive operator should already be in sorted order based on the key used to merge and sequence them together.

### 042_dynamic_import_export_api_at_work

This example shows how to use the SPL APIs for dynamically importing and exporting streams. This is achieved by changing the import and export properties on the fly. This powerful feature in Streams provides a way to change the streams producing and consuming operators to change the way in which they publish and subscribe to streams while the application is running.

### 043_import_export_filter_at_work

This example shows how to use the SPL feature to apply a filter for what gets exported and what gets imported. This powerful feature lets the downstream import operators to specify what kind of tuples they want to receive by specifying conditional expressions involving tuple attributes. That lets the Streams runtime to apply content-based filtering at the point of export. Those who need such a feature to control what information should be sent downstream based on the tuple contents can make use of this flexible feature. This can be done on the fly without stopping and restarting the application.

### 044_streams_checkpointing_at_work

This example shows a key feature of Streams by which an operator's state variables can be preserved when a PE fails and gets restarted. This is done through a combination of the SPL configuration directives named "checkpointing" and "restartable". Developers can protect their critical operator data by taking advantage of this built-in checkpointing feature. When you run this example, you will see data flows without any gaps or interruption, when a PE is killed manually and then gets restored automatically by the Streams runtime.

### 045_file_source_using_spl_custom_operator

This example shows how to create source operators using the Custom operator available in the SPL standard toolkit. Starting in Streams 3.x, it is possible to create source operators without writing primitive source operators in C++ or Java. Simple source operators can be written using the built-in SPL Custom operator. This will come handy for those who don't want to do an extra layer of C++ or Java code for satisfying simple needs for a source operator. You will see a function of a file source operator being implemented all using SPL code in this example.

### 046_launching_external_apps_in_spl

This example shows how to launch/execute an external application within the Streams SPL code. In this case, we defined a simple C++ native function in which we have the required C++ code to launch an external application. That C++ code uses pipes to execute a given application. This function would be useful to launch any custom script within the Streams application logic when certain application specific conditions arise.

### 047_streams_host_tags_at_work

This example shows how to create host tags for a given Streams instance and then use those host tags inside an SPL application. By using host tags, it is possible to avoid hard-coding the host names inside the SPL application code. Detailed instructions about creating and using host tags are explained in this example.

### 048_source_operator_with_control_port

This example shows a way to create a C++ primitive source operator and then provide a control input port for it. Certain classes of applications can make use of this facility to control the kind of data a source operator generates. In addition, this example shows how to pass one or more string literals to the C++ primitive operator as invocation time parameters. As a bonus, this example also shows a simple way to do performance measurement inside the SPL code using the built-in SPL high precision timestamp functions.

### 049_json_to_tuple_to_json_using_java

This example shows how an SPL application can consume JSON formatted data and convert it to SPL tuples. It also shows how to do the reverse action i.e. converting SPL tuples to JSON formatted data. JSON<-->Tuple bidirectional conversion is accomplished via two Java primitive operators that make use of the JSON (Java) libraries shipped as part of the Streams product. Those two Java operators are JSONToTuple and TupleToJSON.

Note: Performance of the JSON<-->Tuple conversion in this example will be limited by the speed of your Java environment. If you want to get better performance, C++ code would help. There is a separate example (055_json_to_tuple_to_json_using_c++) that shows how to do this conversion using C++.

### 050_recursive_dir_scan

This example shows how to use the Streams C++ native function facility to recursively scan a given directory and obtain the names of the files present. The logic for the recursive directory scan polls the specified directory periodically and notifies the downstream operator with a new file that just appeared. There is a companion C++ project for this SPL project. Please refer to the RecursiveDirScanLib project for the C++ logic.

Important sequence of logic for this application:

1) SPL code resolves the C++ native function in its native.function/function.xml file.
2) A call from the SPL code to the native function lands in the wrapper inline C++ function defined in the RecursiveDirScanWrappers.h file of the companion C++ project.
3) From that wrapper function, it gets access to a singleton C++ object of the RecursiveDirScan class and then invokes the getFileNamesInDirectory C++ method.
4) When that C++ method returns, it will have the results stored in a list<string> reference that was passed to it.
5) Back in the SPL code, there is additional logic to cache the already seen files and to filter only the newly found files to send to the downstream operator.

In order to test this application, please refer to the commentary at the top of the SPL file in this project.

*[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED RecursiveDirScanLib THAT IS DESCRIBED BELOW.]*

### 051_native_functions_with_collection_types

This example shows an important feature of Streams. In Streams applications, it may be necessary to accept and return collection types in and out of the C++ native functions. This will require native function code that can directly deal with types such as list, map, and tuple. Streams provides C++ reflection APIs to directly deal with such collection types. In this example, developers can learn how to build native functions inside of a C++ class and then pass list, map, and tuple types to those native functions. In order to run this example, please follow the instructions specified in the README.txt file in the SPL project directory.

*[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED NativeFunctionsWithCollectionTypesLib THAT IS DESCRIBED BELOW.]*

### 052_streams_to_python

This example shows a powerful feature of Streams to wrap existing code assets written using the Python programming language. This example teaches developers how to use the Streams C++ native functions to call any arbitrary Python function and return the results back to SPL code. In order to run this example, please follow the instructions specified in the README.txt file in the SPL project directory. You can also read a very detailed IBM developerWorks technical article about this example:

http://tinyurl.com/c3s56fq

*[THIS EXAMPLE HAS A COMPANION C++ PROJECT CALLED StreamsToPythonLib THAT IS DESCRIBED BELOW.]*

### 053_java_primitive_operator_with_complex_output_tuple_types

This example shows important features that can be done via a Java primitive operator. It shows how to do tracing and logging inside a Java operator. It also shows how we can create an output tuple inside a Java primitive operator to have a list of tuple objects carrying complex typed attributes.

*[THIS EXAMPLE HAS A COMPANION JAVA PROJECT CALLED Java_Complex_Tuple_Type_Submission THAT IS DESCRIBED BELOW.]*

### 054_serialize_deserialize_tuples

This example shows a simple mechanism to serialize and deserialize SPL tuples. When large tuples need to be cached inside an operator's state variable, it will consume a lot of memory. To cut down the memory consumption, one possible option is to convert the large tuples into compact blobs through tuple serialization. When the actual tuple representation is needed, blob values can be converted back to the actual tuples through tuple deserialization. This same method can also be used to transmit data as blobs across operators. This serialization/deserialization approach is likely to add extra CPU overhead. If that extra overhead can be tolerated, then this approach would serve to be useful.

### 055_json_to_tuple_to_json_using_c++

This example shows how an SPL application can consume JSON formatted data and convert it to SPL tuples. It also shows how to do the reverse action i.e. converting SPL tuples to JSON formatted data. JSON<-->Tuple bidirectional conversion is accomplished using an open source C++ JSON API. In order to run this application, you will be required to download an open source component that carries a BSD license. Please read the detailed instructions available in the SPL file for this project. There is also another SPL project that does similar conversion using Java (049_json_to_tuple_to_json_using_java).

### 056_data_sharing_between_fused_spl_custom_and_cpp_primitive_operators

This example shows a particular implementation about how data can be shared across multiple FUSED operators using an SPL map based in-memory store. Here, we are simply showing a way to use the SPL native function facility to perform data sharing via an SPL map based in-memory store that will serve multiple SPL standard toolkit operators and C++ primitive operators. As mentioned above, this example shows data sharing between multiple operators that are fused inside a single PE (Processing Element). This technical approach is called Process Store (ps). This data sharing mechanism will NOT work between operators that are on different PEs. This example depends on the com.ibm.streamsx.ps toolkit that is packaged along with these examples.

### 057_reading_nested_tuple_data_via_file_source

This example shows how to ingest nested tuple data via input files specified in a CSV format. There are certain syntactical rules that need to be followed in specifying data for nested tuples inside a CSV formatted input file. This example is a good one for developers to get an idea about how to do this.

### 058_data_sharing_between_non_fused_spl_custom_and_cpp_primitive_operators

This example is a very useful one that addresses an important aspect of distributed data sharing between Streams built-in and C++ primitive operators that are NOT fused with each other. This technical approach is called Distributed Process Store (dps). It lets the Streams developers share any arbitrarily structured data between multiple PEs (Processing Elements) running on a single or multiple machines. This is accomplished via SPL native functions and with the power of the memcached, redis, cassandra, cloudant, hbase, mongo, couchbase, aerospike and redis-cluster open source data stores. It provides a complementary function to what is already done by the Process Store (ps) in another example above (056_XXXXX). This example depends on the com.ibm.streamsx.dps toolkit that is packaged along with these examples.

### 059_dynamic_scaleout_of_streams_application

This example shows a particular style of writing Streams applications that can be scaled up or scaled down as the application input workload changes. It uses a familiar scenario from the Financial Services Sector, where the price calculation engines will require scaling up when the market data load increases. Code written in this example uses a pattern for starting more instances of an analytic operator to increase parallelism. New instances of such analytic operators can be started on demand without disrupting the already running application flow. As soon as the newly started operator instances are ready, application load will be promptly distributed across the existing and the newly started instances of that operator. In the same way, when the application data load is not high, some of the most recently started operator instances can be stopped to release the CPU cores for other use. This technique is one of many ways to design Streams applications that will scale up and down dynamically according to the changing input data workload.

### 060_simple_pe_failover_technique_at_work

This example shows a way to protect the logic in an analytic operator when its PE (Processing Element) or its host machine crashes. It uses a well-known fail-over technique that is done through a primary/secondary pair configured for an operator that will need safety from PE or machine crash. This example outlines a scheme for protecting the analytic logic written inside an SPL Custom operator against failures. When such failures occur, a specific fail-over technique employed here will continue the business logic without any interruption. This is done by making a secondary PE to takeover the tasks of the failed primary PE. Thus, the secondary PE does the detection of the primary PE's failure and then changes its role from a secondary PE to a new primary PE. All of this is done without losing any data during the fail-over. At the same time, the failed primary PE will be automatically restarted to do its work as a new secondary PE. This particular fail-over technique ensures that there is always a primary/secondary pair working in concert to provide high availability for a business-critical operator that is coded and configured in this manner.

### 061_data_sharing_between_non_fused_spl_custom_operators_and_a_native_function

This example is a very useful one that addresses an important aspect of distributed data sharing between Streams built-in operators that are not fused with each other and a C++ native function. This technical approach is called Distributed Process Store (dps). It lets the Streams developers share any arbitrarily structured data between multiple PEs (Processing Elements) running on a single or multiple machines. This is accomplished via SPL native functions and with the power of the memcached, redis, cassandra, cloudant, hbase, mongo, couchbase,

aerospike and redis-cluster open source data stores. It provides a complementary function to what is already done by the Process Store (ps) in another example above (056_XXXXX). This example depends on the com.ibm.streamsx.dps toolkit that is packaged along with these examples.

### 062_data_sharing_between_non_fused_spl_custom_and_java_primitive_operators

This example is a very useful one that addresses an important aspect of distributed data sharing between Streams built-in operators and a Java primitive operator that are not fused with each other. This technical approach is called Distributed Process Store (dps). It lets the Streams developers share any arbitrarily structured data between multiple PEs (Processing Elements) running on a single or multiple machines. This is accomplished via SPL native functions and with the power of the memcached, redis, cassandra, cloudant, hbase, mongo, couchbase, aerospike and redis-cluster open source data stores. It provides a complementary function to what is already done by the Process Store (ps) in another example above (056_XXXXX). This example depends on the com.ibm.streamsx.dps toolkit that is packaged along with these examples. In this SPL project, you will find a Java primitive operator that exercises all the dps APIs in a very comprehensive manner. In order to get access to the dps APIs, this project's build path is added with dps-helper.jar available inside the com.ibm.streamsx.dps toolkit directory (i.e. impl/java/bin). Please read at the top of this project's SPL file and the TickerIdGenerator.java primitive operator file for an extensive commentary about how to run this example.

### 063_on_the_fly_tuple_creation_and_encoding_decoding_in_java_primitive_operators

This example shows how to create a tuple on the fly inside a Java primitive operator. In addition, this example also shows how to convert a tuple into a blob (Java byte buffer) and how to convert a blob (Java byte buffer) in to a tuple. It is an interesting concept that a Java primitive operator developer can put into use in certain situations that warrant dynamic tuple creation, tuple encoding and decoding all inside Java.

### 064_using_spl_composite_params

This example shows different ways in which parameters can be passed to SPL composites. It is very useful to pass parameters as attributes, expressions, functions, operators, and types. These different ways of passing parameters to the composites is the focus of this example.

### 065_using_multiple_threads_in_java_operator

This example shows how to spawn multiple threads within a Java primitive operator and then submit tuples from within those threads concurrently.

### 066_load_balancing_using_gate

As documented in the Streams Info Center for a ThreadedSplit, if the processing time of a tuple varies considerably depending on the tuple data, it may cause problems where a tuple with a long processing time may cause subsequent tuples to be backed up in the stream. This example shows how a Gate operator can be combined with the ThreadedSplit can be used to ensure load balancing.

### 067_simple_java_source_operator

This example shows a basic source operator implemented in Java. There are specific steps required for implementing a source operator and it can be learned in this example.

### 068_tuple_introspection_inside_java_operator

This example shows how a tuple can be introspected to learn about its structure and its attribute names and their types. Inside a Java operator, this example illustrates how it is possible to recursively look through a tuple to understand its composition.

### 069_changing_map_value_during_iteration

Until the release of Streams version 3.2.1, it was not possible to modify the value of a map inside an iteration loop. This example shows a new feature available in Streams version 3.2.1 that permits the value of a map to be modified inside a for loop.

### 070_convert_block_data_into_tuples_using_parse

This example shows how a block of data ingested as a blob type can be converted into individual tuples using the Parse operator.

### 071_java_native_functions

Java native functions provide a cool way to add user-defined functions in Java and then call them directly within the SPL code. This example shows how easy it is to create java native functions.

### 072_using_streams_rest_apis

Streams provides REST APIs to query different kinds of metrics about the instances, jobs, resources during the runtime operation. It is a comprehensive set of APIs that can be used with proper security configuration. This example shows a few different REST APIs in action by invoking them within Java code.

### 073_java_operator_fusion

This example shows how two different Java operators one performing the Sink operation and the other performing the analytics operation can be fused to operate within a single PE.

### 074_user_defined_parallelism_01 - 085_user_defined_parallelism_12

User Defined Parallelism (UDP) is an excellent feature introduced in Streams version 3.2 and it provides a very simple way to run either a part or a full flow graph in a concurrent manner. It is an easy mechanism to scale and speed up the Streams flow graph. These 12 different examples provide a walk through of various UDP scenarios.

### 086_jms_source_sink_using_activemq

This example shows how the JMSSource and JMSSink operators from the Streams standard toolkit can be put to use for sending messages from Streams into the Apache ActiveMQ queues and topics as well as reading messages from there into Streams.

### 087_email_alerts_via_java_native_function

This example shows a way to send email alerts from an SPL application. It is done via a Java native function by using the email API available in the standard Java platform. If an SMTP server is present in the same network where Streams servers are connected, the technique shown in this example can be put to use for sending email alerts.

### 088_java_operator_params_and_multiple_input_output_ports

This example demonstrates two different features of the Java primitive operator framework. It first shows how operator parameters can be easily processed inside the Java operators via the @Parameter annotations. Then, it shows how multiple input and output ports can be accessed inside the Java operators. As a bonus, it also shows a better approach for on the fly creation of the output tuples made with complex nested types.

### 089_integrating_streams_apps_with_web_apps

This example demonstrates one of the Streams open source toolkits (com.ibm.streamsx.inet). Using this toolkit one can integrate Streams applications with web applications. Please read the comments in the SPL file for this example project to download that toolkit, install it, and then use that toolkit inside a simple SPL application.

### 090_consistent_region_spl_01

This example demonstrates how a consistent region can be defined for the entire application topology starting from a Beacon with an operator driven checkpoint trigger. One of the operators in this application is forcefully aborted inside the application to prove that application

will continue processing tuples normally after an automatic restart of that failed operator.

### 091_consistent_region_spl_02

This example demonstrates how a consistent region can be defined for the entire application topology starting from a FileSource with a periodic checkpoint trigger. One of the operators in this application is forcefully aborted inside the application to prove that application will continue processing tuples normally after an automatic restart of that failed operator.

### 092_consistent_region_spl_03

This example demonstrates how a consistent region can be defined for the entire application topology starting from a Beacon with an operator driven checkpoint trigger. One of the Aggregate operators in this application is forcefully aborted inside the application multiple times to prove that application survive those multiple crashes at different times and yet will continue processing tuples normally after an automatic restart of that failed operator. In addition, during those crashes Streams will preserve the windows contents of that Aggregate operator.

### 093_consistent_region_spl_04

This example demonstrates how a consistent region can be defined for two different composites acting as sources for this application. These consistent regions have a periodic checkpoint trigger. Couple of different Custom operators connected to those sources are forcefully aborted inside the application. Output streams of those operators will be combined using a Join operator. This application will ensure that the application will continue normally without losing any tuples by withstanding the random crash of those two Custom operators.

### 094_consistent_region_spl_05

This particular example shows how only a portion of the topology will take part in the consistent region by having an autonomous section in the application graph. This example simulates the operator failure by aborting one of the operators automatically when the application is in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed and the Join operator's window state will not be compromised during the course of the unexpected operator failure and the subsequent recovery/restoration. At the same time, parts of the application that is in the autonomous area will get duplicate tuples during a crash recovery happening in the consistent region of this application graph. This example's purpose is to make the

users aware of this fact. In the autonomous area, measures need to be taken to do deduplication.

### 095_consistent_region_spl_06

This particular example shows how a non-replay capable Source operator will not be a show stopper when it comes to employing the consistent region feature in such applications. When using sources (such as TCPSource) that can't realistically replay data, there is way to configure your application with consistent region by using an utility operator called ReplaybleStart (shipped with the Streams product). In this example, we will use a topology that uses TCPSource along with ReplayableStart to achieve application-level fault tolerance. This example simulates the operator failure by aborting one of the operators automatically when the application is in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed and the Aggregate operator's window state will not be compromised during the course of the unexpected operator failure and the subsequent recovery/restoration.

### 096_consistent_region_spl_07

This particular example shows how a C++ primitive operator can play a role inside a consistent region. Please look at the CPP interface (.h) and the implementation (.cpp) files inside the SimpleSourceOp sub-directory in this SPL project. There are certain callback functions that the C++ operator developer needs to implement the checkpoint and restore state events. This example simulates the operator failure by aborting one of the operators in the flow graph automatically when the application is in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed during the course of the unexpected operator failure and the subsequent recovery/restoration.

### 097_consistent_region_spl_08

This particular example shows how a C++ primitive operator can play a role inside a consistent region. Please look at the CPP interface (.h) and the implementation (.cpp) files certain callback functions that the C++ operator developer needs to implement the checkpoint and restore state events. This example simulates the operator failure by aborting one of the operators in the flow graph automatically when the application is in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed and the

application state kept inside this C++ operator will be preserved during the course of the unexpected operator failure and the subsequent recovery/restoration.

### 098_consistent_region_spl_09

This particular example shows how a Java primitive operator can play a role inside a consistent region. Please look at the Java file in the impl/java/src sub-directory in this SPL project. There are certain callback functions that the Java operator developer needs to implement the checkpoint and restore state events. This example simulates the operator failure by aborting one of the operators in the flow graph automatically when the application is  in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed during the course of the unexpected operator failure and the subsequent recovery/restoration.

### 099_consistent_region_spl_10

This particular example shows how a Java primitive operator can play a role inside a consistent region. Please look at the Java operator code inside the impl/java/src sub-directory in this SPL project. There are certain callback functions that the Java operator developer needs to implement the checkpoint and restore state events. This example simulates the operator failure by aborting one of the operators in the flow graph automatically when the application is in the middle of executing the logic. By doing that, the core fault tolerance feature of the consistent region will get triggered to recover from a failure that occurred in an application graph. It will prove that the tuples will not be missed and the application state kept inside this Java operator will be preserved during the course of the unexpected operator failure and the subsequent recovery/restoration.

### 100_using_jmx_api_01

This is a plain Java application written to show how one can use the JMX APIs available in Streams 4.x and higher versions. Using the JMX (Java Management Extensions) APIs, it is convenient to monitor and manage Streams artifacts. This example shows how one can use the Streams JMX APIs to query information about the Streams domain and the Streams instance.

### 101_using_jmx_api_02

This is a plain Java application written to show how one can use the JMX APIs available in Streams 4.x and higher versions. Using the JMX (Java Management Extensions) APIs, it is convenient to monitor and manage Streams artifacts. This example shows how one can use the

Streams JMX APIs to fetch the bulk contents from a log file for a given domain.

### 102_using_jmx_api_03

This is a plain Java application written to show how one can use the JMX APIs available in Streams 4.x and higher versions. Using the JMX (Java Management Extensions) APIs, it is convenient to monitor and manage Streams artifacts. This example shows how one can use the Streams JMX API notifications to get alerted via callback functions about an inactivity timeout in a given Streams domain.

### 103_view_annotation_at_work

This is a simple SPL application that explains the steps required to use the view annotation and then how to visualize the view annotated stream in the Streams web console. Detailed steps to view the annotated stream are shown in the commentary section of this SPL file.

### com.ibm.streamsx.ps

This is an SPL toolkit that contains the C++ source code for the Process Store (ps) implementation. This toolkit can be used to share data between different operators that are fused into a single PE. You can find the source code in the impl/include and in the impl/src sub-directories. You will also find the pre-built .so file from the C++ logic of this toolkit inside the impl/lib sub-directory for RHEL5, CentOS5, RHEL6 and CentOS6 flavors of Linux. This toolkit also includes a test application that shows how to use many of the available ps APIs. More details about this toolkit can be found in doc/ps-usage-tips.txt file.

### com.ibm.streamsx.dps

This is an SPL toolkit that contains the C++ source code for the Distributed Process Store (dps) implementation. This toolkit can be used to share data between different PEs running on the same or different machines. You can find the full source code in the impl/include and in the impl/src sub-directories. APIs to access the dps can be called literally from anywhere inside a Streams application. To fulfill that idea, there is a Java layer (impl/java/src) that will expose the dps APIs to Java primitive operators. There is a jar file in the impl/java/bin that must be added to the build path of a Java primitive operator project for using the dps APIs. You will also find the pre-built .so file from the C++ logic of this toolkit inside the impl/lib sub-directory for RHEL5, CentOS5, RHEL6, CentOS6 and IBM Power (ppc64) flavors of Linux. The dps functions are achieved using a back-end data store (memcached or redis or cassandra or cloudant or hbase or mongo or couchbase or aerospike or redis-cluster). There are pre-built client libraries in the imp/lib sub-directory to readily access memcached, redis, cassandra, cloudant, hbase, mongo, couchbase,

aerospike and redis-cluster servers. This toolkit also includes a test application that shows how to use many of the available dps APIs. More details about this toolkit can be found in doc/dps-usage-tips.txt file.

---

Use the Eclipse Java perspective to work with the following projects.

### RSS_Reader_Standalone

This is a stand-alone Java project that encapsulates the RSS reader function. Before the RSS reader logic can be added to an SPL Java primitive operator, this stand-alone Java project is used to verify and test the business logic. Once its functionally is working, the core logic is ready to be moved into the process function of an SPL Java primitive operator.

### RSS_Reader_Primitive

This is a companion Java project for the SPL project (033_java_primitive_operator_at_work) present in the same Eclipse workspace. This Java class extends from a Streams AbstractOperator class. It receives an input tuple with information about an RSS provider. Then, it queries that RSS server for the currently available RSS feeds. It parses the RSS XML data and converts that into an output tuple and sends it via its output port.

### Java_Complex_Tuple_Type_Submission

This is a companion Java project for the SPL project (053_java_primitive_operator_with_complex_output_tuple_types). This Java primitive operator answers the following questions.

1) How can we ingest an input tuple with attributes of different types and access those tuple attributes inside a Java operator?
2) How can we write to the PE trace file from a Java operator?
3) How can we write to the PE log file from a Java operator?
4) How can we write to the PE console output file from a Java operator?
5) More importantly, how can we create an output tuple containing a list of complex typed attributes and then submit it inside a Java operator?

---

Use the Eclipse CDT perspective to work with the following projects.

### NativeFunctionLib

This is a companion project for the 032_native_function_at_work SPL project present in the same Eclipse workspace. This C++ project gets compiled into a shared library (.so) file so that an SPL project can access the native C++ functions available in the .so file. In the SPL project, there is a native function model that will define the location of the .so file, C++ function name and the C++ namespace. Before

working on the SPL project, it is necessary to run the mk script in the C++ project directory. Run this script from a terminal window and it will copy the C++ include file, CPP file, and the .so files into the impl directory of the SPL project.

### PrimitiveOperatorLib

This is a companion project for the 036_shared_lib_primitve_operator_at_work SPL project present in the same Eclipse workspace. This C++ project gets compiled into a shared library (.so) file so that the SPL C++ primitve operator can access the C++ functions available in the .so file. In the SPL project, there will be an operator model that will define the location of the .so file, C++ function name and the C++ namespace. Before working on the SPL project, it is necessary to run the mk script in the C++ project directory. Run this script from a terminal window and it will copy the C++ include file, CPP file, and the .so files into the impl directory of the SPL project.

### RecursiveDirScanLib

This is a companion project for the 050_recursive_dir_scan SPL project. Before using that SPL project, it is required to build this C++ companion project from a terminal window. Simply run the mk script available in this project directory. It will build a shared object (.so) file and copy it to the impl/lib directory of the SPL project. This C++ project has one CPP file and two include files. There is a wrapper include file that gives an entry point from the SPL directly into the C++ native function code. Browse those three source files to understand further.

### NativeFunctionsWithCollectionTypesLib

This is a companion project for the 051_native_functions_with_collection_types SPL project. Before using that SPL project, it is required to build this C++ companion project from a terminal window. Simply run the mk script available in this project directory. It will build a shared object (.so) file and copy it to the impl/lib directory of the SPL project. This C++ project has one CPP file and two include files. There is a wrapper include file that gives an entry point from the SPL directly into the C++ native function code. Browse those three source files to understand how you can pass SPL collection types to the C++ native functions.

### StreamsToPythonLib

This is a companion project for the 052_streams_to_python SPL project. Before using that SPL project, it is required to build this C++ companion project from a terminal window. Simply run the mk script available in this project directory. It will build a shared object

(.so) file and copy it to the impl/lib directory of the SPL project. Please refer to this URL for more details:
http://tinyurl.com/c3s56fq

**UsePrimitiveOperatorLib**
This is a client a.k.a test driver program that uses a shared library. This stand-alone test application calls a function available inside the PrimitiveOperatorLib shared (.so) library. Before using the shared library inside an SPL C++ primitive operator, it is recommended to test it first using a test driver.

In Eclipse CDT, there are specific project build properties for a test driver program to link with the .so library. Refer to the commentary at the top of the test driver CPP file.

---

The following SPL examples are directly taken out of the SPL Introductory Tutorial PDF file. They are also added to the mix inside the same tar.gz file. This package containing a total of 122 examples is targeted to assist the newbie enthusiasts of the InfoSphere Streams SPL language. Please refer to the SPL Tutorial PDF file for a detailed description about the applications listed below.

**901_cat_example**
**902_word_count**
**903_unique**
**904_primitive_round_robin_split**
**905_gate_load_balancer**

---

A detailed procedure to import the "SPL-Examples-For-Beginners" into Streams Studio is described below.

1) On a 64 bit Redhat Enterprise Linux 5.8 or above, download and unzip the file "SPL-Examples-For-Beginners.tar.gz".
(URL: http://tinyurl.com/3apa97q)

tar -xvzf SPL-Examples-For-Beginners.tar.gz

2) Install the following in your home directory.

a) 64 bit version of InfoSphere Streams
b) Using the First Steps Streams application, install Streams Studio.

3) Start the Streams Studio with a workspace location pointing to the directory where you unzipped in step (1) above (e-g: ~/SPL-Examples-For-Beginners).

4) In the left pane, select the "Streams Explorer" tab.

5) Right click on "Toolkit Locations" and select "Add Toolkit Location".

6) In the resulting dialog box, click the "Directory" button.

7) In the resulting dialog box, navigate until you can see the "<Your_Streams_Install_Directory>/toolkits/com.ibm.streams.db" directory.

8) Select "com.ibm.streams.db" and click OK twice. In the same way, you also have to add the messaging toolkit directory from "<Your_Streams_Install_Directory>/toolkits/com.ibm.streams.messaging". After this, on the left pane, select the "Project Explorer" tab.

If you are using Streams Studio version 3.1 or earlier, skip to step 9 now. If you are using Streams Studio version 3.2 or later, you may have to make this SPL build configuration changes inside Streams Studio to avoid SPL import and compiler errors.

a) Click on the Window-->Preferences menu in your Streams Studio.
b) In the resulting dialog box, on the left pane select InfoSphere Streams-->SPL Build.
c) In the right panel, ensure that all the checkboxes are unselected.
d) In the "Toolkit lookup paths" setting, ensure that "Include direct/indirect toolkits only" option is selected.
e) Click Apply and then click OK.

9) Let us now import the "SPL-Examples-For-Beginners" project artifacts. Select "File->Import".

10) In the resulting dialog box, expand "General" and select "Existing Projects into Workspace". Click Next.

11) In the next dialog box, click on "Select root directory" and then click on the "Browse" button.

12) Navigate until you can see the "SPL-Examples-For-Beginners" directory that you extracted in step 1.

13) Now, select the "SPL-Examples-For-Beginners" directory and click OK.

14) In the next dialog box, all projects in the "SPL-Examples For-Beginners" directory should be selected.

15) Ensure that an option named "Copy projects into workspace" is **NOT** selected (It is located below the list of projects).

16) Click "Finish".

17) You should now see all the SPL, Java, and C++ projects getting imported into your workspace.

18) Shortly after the import finishes, all the SPL projects a.k.a toolkits will be automatically indexed and built. You can watch the "SPL Build" console to monitor the progress of the SPL projects that are being built.

19) Approximately after two hours, all the SPL and Java projects will be in a "Fully built" state.

20) However, some of the SPL projects will show build problems with red error markers. Such SPL projects are typically those with a C++ companion project.

Let us resolve the build errors.

a) Open a Linux terminal window.
b) Change directory to each of the five C++ projects one at a time. (NativeFunctionLib, NativeFunctionsWithCollectionTypesLib, PrimitiveOperatorLib, RecursiveDirScanLib, StreamsToPythonLib)
c) Inside every C++ project's directory, execute the library build script by typing this command: ./mk
d) After completing step (c) for all the C++ projects, you can right-click on those SPL projects with red error markers and select "Build Active Configurations".
e) Step (d) above should have resolved all the build errors.
f) You may still see a red error marker for the SPL project named "027_java_op_at_work". That is a Java related error and we can live with that, as it will not affect the functionality of that project.
g) If you see any errors in the projects containing a Java Primitive operator, usually that means you have Java build path issues. Please open the Java Primitive Operator source file in that project. Then, read the comments at the top of that source file and resolve the errors. This involves a task to right-click on the project name and then to select properties. In the resulting dialog box, select "Java Build Path" in the left pane. Then, select the "Libraries" tab in the right pane. You will see a few jar files pointing to wrong directory names. You can delete them and press "Add external jars" and then navigate to the correct directory on your machine and then select the correct jar file.

h) For the StreamsToPythonLib C++ project, you have do this if the include path is not configured already. Add an include path "/usr/include/python2.6" in the "GCC C++ Compiler"->"Includes" section. You can do this by right clicking on that project name and then by selecting "Properties". In the left pane of the resulting dialog, expand "C/C++ Build" and then select "Settings" and then proceed to add the include path as explained above.

i) IMPORTANT: If you are using Streams Studio 3.2 or later, you may have red error markers for the 053_XXXX, 062_XXXX, 098_XXXX to 102_XXXX projects. You must follow the instructions specified at the top of the SPL file in those projects and resolve the errors to ensure that the red error markers are gone. If you see more red markers in other projects (056_XXXX, 058_XXXX etc.), right-click on those projects and select "Build Active Configurations" to force a build.

21) At this point, you have imported all 122 examples into your Streams Studio. What is next? It is time now to immerse in Streams. A popular spot to begin your deep dive is 001_hello_world_in_spl project!!!