

HJC SOLUTIONS LTD



Canyon User Guide

The flat file processing framework

<http://canyon.hjcsolutions.co.uk>

Table of Contents

Contents

Introduction	3
Why use flat files.....	3
Chapter 1 - Getting Started.....	5
5 Steps to integration.....	5
Step 1: Create Java model beans	5
Step 2: Create Canyon mapping files	6
Step 3: Configure Canyon	6
Step 4: Add Canyon iterator to your code	7
Chapter 2 - Canyon configuration.....	8
Mapping Elements	9
config (REQUIRED)	9
input (OPTIONAL if output set)	9
output (OPTIONAL if input set)	10
Properties.....	10
Chapter 3 - Class Mappings.....	12
Mapping element.....	12
Class Definition	12
Attribute types.....	13
Id Mapping	13
Property Mapping	14
Object Mapping	15
Collection Mapping	15
Set Mapping	15
Chapter 4 - Relationships and keys.....	17

Introduction

If you need to read data from a flat file for processing in Java then Canyon is for you.

Canyon is a lightweight, flexible and high performing flat file mapping framework. It has been designed to allow the quick and easy mapping of different formats of text based flat files into JAVA objects. Much of the Canyon framework is based on the Hibernate way of doing things, and bringing much of that experience to the flat file world.

In Canyon terms a flat file is any text file that either has a delimited field, or a fixed length field format. Canyon also expects each record to exist on a single line. Multi line records are currently not supported.

Canyon uses mappings and configurations very similar to Hibernate to bind data found in flat files into Java objects. The mapping information is completely separate from the class or file information. In fact the same file can be used by multiple mappings at the same time to represent different views of the same data.

Why use flat files

In many enterprise architectures the database is king, and quite rightly so. It provides a robust, structured mechanism for persistence and retrieval of data. However in some cases a database can become a hindrance rather than a friend. Instances include managing huge datasets, transferring that data set to multiple sources. In these cases flat files rule.

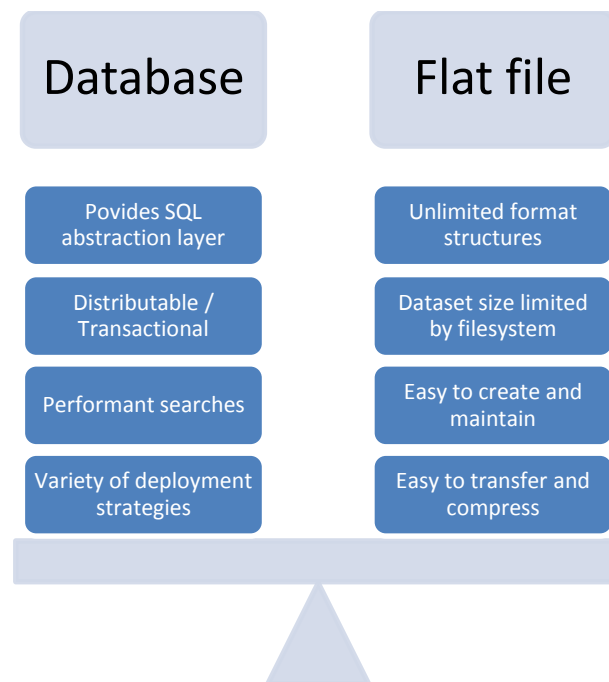


Figure 1

Flat files are the most efficient way of representing data because the format and structure of that data is implicitly known by the provider and the consumer. Flat files also allow easy transfer and compression. However you lose the easy ability to access the data as you would from a database. Canyon aims to step into this breach.

In this guide we provide a quick example that shows a lot of what Canyon has to offer. Followed by a bit more detail in how to configure and integrate Canyon into your solution.

Chapter 1 - Getting Started

To get you up to speed using Canyon we will be using an example to demonstrate some of the features. The sample will be based on three files being mapped to three classes. This is not mandatory but it makes your mapping files straight forward. Essentially you could think of each file as a database table.

5 Steps to integration

There are 5 steps to configuring Canyon

1. Create your JAVA model. Canyon requires your model follow the Java Bean convention, i.e a default no args constructor, and getter/setter methods for the attributes.
2. Create the mapping file for each object model
3. Create the canyon.cfg.xml file to combine the mapping files with the physical files or sources. Place it in your root classpath (similar to log4j, hibernate etc)
4. Embed a SessionFactory and Session object in your code and iterate over the objects
5. Done!

The best way to get you started with Canyon is by example. The tutorial application, which is an Eclipse project, can be found at <http://canyon.hjcsolutions.co.uk/downloads>.

Step 1: Create Java model beans

The model for the tutorial is below

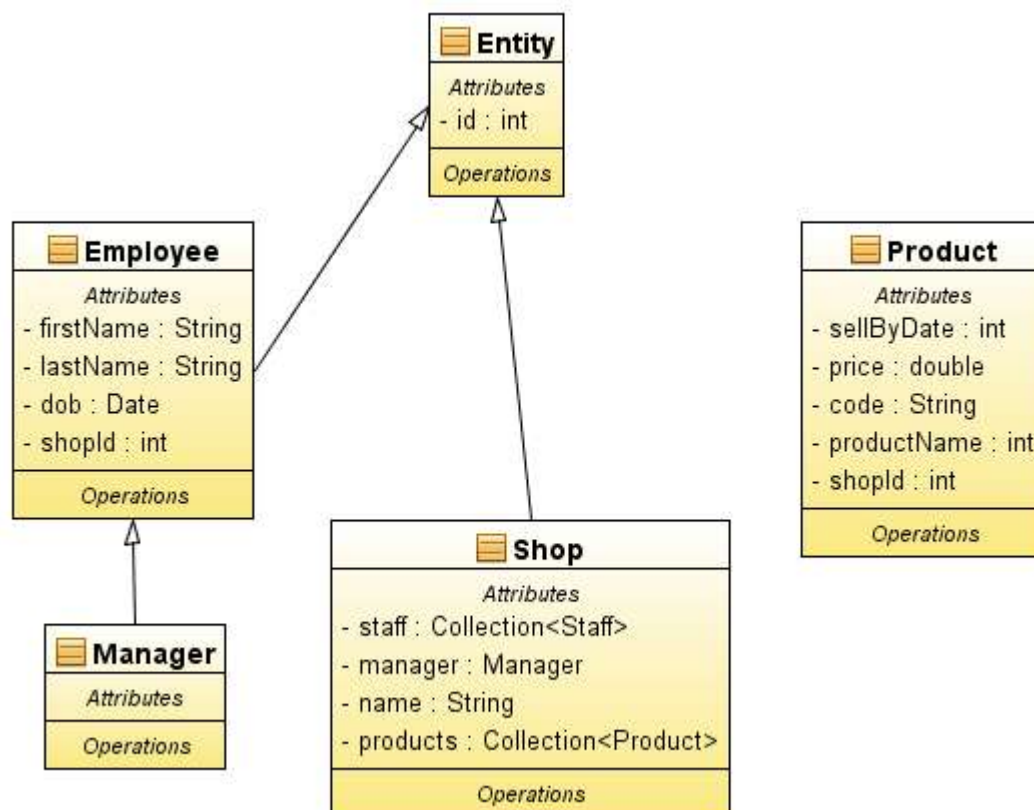


Figure 2

Step 2: Create Canyon mapping files

The first thing to do is to map your input files to Java objects using a Canyon mapping file. The

DTD to a file is here http://canyon.hjcsolutions.co.uk/canyon_mapping.dtd

The mapping for the shop and product object is below. As you can see the shop uses a delimited file, whereas the products use a fixed length field file. The other mapping files can be found in the project archive.

```
<!DOCTYPE mapping SYSTEM "http://canyon.hjcsolutions.co.uk/canyon_mapping.dtd">
<mapping>
  <class name="co.uk.hjc.canyon.example.Shop" type="DELIMITED" delimiter=",">
    <id name="id" position="0" type="integer"/>
    <property name="name" position="1" type="string" trim="true"/>
    <object name="manager" type="co.uk.hjc.canyon.example.Manager" join="shopId"/>
    <collection name="products" type="co.uk.hjc.canyon.example.Product"
      join="shopId"/>
    <set name="staff" type="co.uk.hjc.canyon.example.Employee" join="shopId"/>
  </class>
</mapping>
```

Config 1 - Example mapping - Shop

```
<!DOCTYPE mapping SYSTEM "http://canyon.hjcsolutions.co.uk/canyon_mapping.dtd">
<mapping>
  <class name="co.uk.hjc.canyon.example.Product" type="FIXED_LENGTH" start_index="1"
    start_line="1">
    <id name="code" start="1" length="10" type="long"/>
    <property name="sellByDate" start="11" length="10" type="date"
      date_format="yyyy-MM-dd"/>
    <property name="price" start="22" length="5" type="double" />
    <property name="productName" start="28" length="20" type="string" trim="true"/>
    <property name="shopId" start="49" length="10" type="long"/>
  </class>
</mapping>
```

Config 2 - Example mapping - Product

Step 3: Configure Canyon

Canyon uses a very simple configuration file to drive its functionality. Essentially the configuration file combines a mapping configuration with a input and/or output file. One or the other is required. The configuration should be named **canyon.cfg.xml** and be placed in the root class path of your application. Canyon will then search for the file. Alternatively you can create an environment variable called **canyon.config_file_location** that specifies the file path of the configuration file.

```
<canyon-configuration>
  <session-factory name="example 1">
    <!-- property name="canyon.allow_default_values" value="true"/-->
    <mapping>
      <config resource="mappings/shop.mapping.xml" />
      <input resource="shops.txt" />
    </mapping>
    <mapping>
      <config resource="mappings/product.mapping.xml" />
      <input resource="products.txt" />
    </mapping>
    <mapping>
      <config resource="mappings/manager.mapping.xml" />
      <input resource="manager.txt" />
    </mapping>
    <mapping>
```

```
        <config resource="mappings/employee.mapping.xml" />
        <input resource="employee.txt" />
    </mapping>
</session-factory>
</canyon-configuration>
```

Config 3 - Example canyon.cfg.xml

As the diagram shows, each mapping combines a mapping document as defined in Step 2 and physical file or resource. In the case of this example only resource(s) are used. Alternatively the *file* attribute could be used, where the value would be the full path to the input file.

Step 4: Add Canyon iterator to your code

With the mappings created, and Canyon configured, you are now ready to add Canyon to your code. This is done extremely simply with the following piece of code.

```
Session session = SessionFactory.getSession("example 1");
while (session.hasNext(Employee.class)) {
    Employee employee = session.next(Employee.class);
    // do something with the employee...
    System.out.println("Processing..." + employee);
}
```

Code 1 - Example integration code

And that is it! Depending on your mapping to the file being accurate, there is nothing more to do other than iterate over your objects. By passing in different objects, the iterator will process differently. E.g if you were to process a shop (in the example there are only two), Canyon would automatically process all the related employees and products. Please see Chapter 4 - for more information on relationships and how they work.

Chapter 2 - Canyon configuration

Configuring Canyon is very straight forward. All that is required is the mapping file for each file to class mapping, and the location of the input/output file.

There are three ways to specify the location of the Canyon configuration file. The first and easiest way is to create a file name canyon.cfg.xml and place it in the root classpath of your application (the same way you would put your hibernate.cfg.xml or your log4.xml file). Canyon will then scan the classpath for the file and load it automatically.

The second way is to use a System property called canyon.config_file_location which specifies the path to the physical file.

The final option is to use the SessionFactory.getSession(String configResourceName) method to obtain a session. The configResourceName can be a class-path location, property name, physical file location or the name of a CanyonConfigProvider. Canyon will attempt to use all four in that specified order.

Diagram 2 shows a sample configuration file

```
<!DOCTYPE mapping SYSTEM "http://canyon.hjcsolutions.co.uk/canyon_config.dtd">
<canyon-configuration>
  <session-factory name="example 1">
    <property name="canyon.close_on_eof" value="false">
    </property>

    <mapping>
      <config resource="mappings/shop.mapping.xml" />
      <input resource="shops.txt" />
    </mapping>

    <mapping>
      <config resource="mappings/product.mapping.xml" />
      <input resource="products.txt" />
    </mapping>

    <mapping>
      <config resource="mappings/employee.mapping.xml" />
      <input resource="employee.txt" />
      <output file="/data/new_employee.txt" />
    </mapping>

    <mapping>
      <config provider="com.example.CustomProvider">
        <property name="custom.propertyname" value="custom.propertyValue">
        </property>
      <config provider="com.example.CustomProvider">
      </config>
      <output file="/data/new_employee.txt" />
    </mapping>

  </session-factory>
</canyon-configuration>
```

The diagram shows that the Canyon configuration requires the definition of a 1 or more session factories. The Session is the main interface into Canyon. Each session factory definition works independently of the other. This allows the possibility to share mappings and even inputs/outputs across sessions (this does create race hazards and non predictable behavior however).

It should also be noted that because Canyon is working directly with files, outside of transactional control Canyon could be used in a non thread safe way. To avoid this, the main methods have been synchronized. However you can opt out and work with an unsynchronised non thread safe Session¹.

A session factory must have a unique name and define one or more mappings. The mapping combines a mapping configuration and an input source and/or output destination.

Canyon allows you to specify the location of any of the files² as streams (resources) or the physical file location.

A session can also set one of number of properties.

Mapping Elements

config (REQUIRED)

Attribute Name	Required	Description
resource	REQUIRED (if not file, property or provider)	the name of the stream that can be found in the classpath
file	REQUIRED (if not resource, property or provider)	the full path of the file to use as the mapping configuration
property	REQUIRED (if not resource, file or provider)	the name of the property that holds the value of the physical location of the configuration file on the system
provider	REQUIRED (if not resource, property or file)	The full class name of the custom provider that will load the input stream. Class must implement the CanyonConfigProvider interface

input (OPTIONAL if output set)

For each mapping you can provide 0 or more input definitions. The inputs will then be parsed in the order they are defined until all streams are exhausted.

Attribute Name	Required	Description
resource	REQUIRED (if not file, property, provider or directory)	the name of the stream that can be found in the classpath
file	REQUIRED (if not resource, property, provider or directory)	the full path of the file to use as the mapping configuration
property	REQUIRED (if not resource, file or provider or directory)	the name of the property that holds the value of the physical location of the stream
provider	REQUIRED (if not resource, property, file or directory)	The full class name of the custom provider that will load the input stream. Class must implement the CanyonInputProvider interface

¹ This can be done by adding the property `canyon.sync_session` and the value `false`. By default the synchronised session is used.

² Currently Canyon only supports file for output

directory	REQUIRED (if not resource, file, property or provider)	The directory to scan for files. All files will be consumed in the order they are returned by File.listFiles().
-----------	--	---

output (OPTIONAL if input set)

For each mapping you can provide 0 or 1 output definition.

Attribute Name	Required	Description
file	REQUIRED (if not property or provider)	the full path of the file to use as the mapping configuration
property	REQUIRED (if not resource, file or provider)	the name of the property that holds the value of the physical location of the stream
provider	REQUIRED (if not resource, property or file)	The full class name of the custom provider that will load the input stream. Class must implement the CanyonOutputProvider interface

Properties

As Canyon matures there may be more properties supported to add additional flexibility. At the time of writing the properties supported are as follows.

Property name	Possible values	Description
canyon.close_on_eof	true false	Tell the session to close all streams once file processing is complete. By default set to true.
canyon.sync_session	true false	Select whether the factory should create an unsynchronised session by setting to false. By default set to true.
canyon.allow_default_values	true false	Select whether default values should be applied if the data retrieved is null. By default this is true.
canyon.default_integer	any integer	The default integer value to use if allow_default_values is true. By default this is 0.
canyon.default_long	any long	The default long value to use if allow_default_values is true. By default this is 0
canyon.default_string	any string	The default string value to use if allow_default_values is true. By default this is "" (empty string)
canyon.default_date	any date with the format	The default date value to use if

	DDMMyyyy (day, month, year)	allow_default_values is true. By default this is the current system date.
canyon.default_boolean	true false	The default boolean value to use if allow_default_values is true. By default this is true
canyon.default_short	0-9	The default integer value to use if allow_default_values is true. By default this is 0
canyon.default_double	any double	The default double value to use if allow_default_values is true. By default this is 0.0

Chapter 3 - Class Mappings

The class mapping mechanism is based loosely on the hibernate format. This was done to make it a familiar interface to using Canyon. Canyon supports any Java bean. This means any Java class with a no arguments constructor and get/setter methods for its managed attributes. This is required because Canyon (as does most frameworks) uses reflection to gain access to the attributes by their name.

Diagram 2 shows a simple example mapping file.

```
<!DOCTYPE mapping SYSTEM "http://canyon.hjcsolutions.co.uk/canyon_mapping.dtd">
<mapping>
  <class name="example.Employee" type="FIXED_LENGTH" start_index="1">
    <id name="id" start="1" length="10" type="long"/>
    <property name="firstName" start="11" length="10" type="string"/>
    <property name="lastName" start="21" length="15" type="string" />
    <property name="dob" start="36" length="10" type="date"
      date_format="yyyy-MM-dd" />
    <property name="shopId" start="46" length="10" type="long"/>
  </class>
</mapping>
```

This diagram shows the main elements of a mapping file.

1. mapping element
2. class definition, and file type definition
3. id and property fields, which are the class attributes that can be persisted

By using the doctype shown above pointing to the DTD most ides will provide code completion as you write your mapping file. It will also validate your file to ensure that it complies with the required data.

Mapping element

The mapping element is just a wrapper around the class definition. A mapping can only hold one class. This is because the mapping of the class is one to one.

Class Definition

A class definition has the following attributes

Attribute Name	Required	Description
name	REQUIRED	the full name of the class, including the package details
type	REQUIRED	The type of mapping. Either FIXED_LENGTH or DELIMITED
start_index	OPTIONAL	For fixed length files what value should the index numbers start at. By default this is 0. However if the details of the mapping start at a different value (commonly 1) then providing it here will override the default
delimiter	OPTIONAL	The character used as the delimiter. By default the delimiter is ','.

start_line	OPTIONAL	The number of lines to skip at the start of processing. This helps in the cases where the top of the file contains header information, or unnecessary data.
------------	----------	---

Attribute types

Canyon out of the box supports different attribute types. They are the standard primitive types, collections, sets and other objects. Canyon also allows the designation of one of the fields to be marked as an identifier. This restriction of only one field being an id has been added to ensure usage and configuration is as easy as possible. It was also decided that the majority of most flat file use cases will not require multiple keys and joins. In the case where it is required the flat file format may not be the best option, and maybe a database should be investigated.

Id Mapping

The id property is a optional property that can be added to a class to enable relationships to be maintained with other Canyon classes. An id can be any primitive attribute on your class that can be used as a key.

Attribute name	Required	Description
name	REQUIRED	the name of the attribute to use as the id
start	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the position this field starts at.
length	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the length of field
type	REQUIRED	The type of the attribute. Can be one of (string, long, short, date, integer)
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the DefaultPropertyAccessor is used.
trim	OPTIONAL	true if the type is string and the value should be trimmed (i.e preceding and proceeding spaces are removed. False if the value should not be trimmer. By default trim is false.
date_format	OPTIONAL	A string describing how the date should be formatted. By default it uses dd-MM-yyyy
position	OPTIONAL	For delimited files an integer explaining which field is to be used

Property Mapping

The property of a class can be any primitive type.

Attribute name	Required	Description
name	REQUIRED	the name of the attribute to use as the id
start	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the position this field starts at.
length	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the length of field
type	REQUIRED	The type of the attribute. Can be one of (string, long, short, date, integer,int, character, char, Boolean, bool)
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the DefaultPropertyAccessor is used.
trim	OPTIONAL	true if the type is string and the value should be trimmed (i.e preceding and proceeding spaces are removed. False if the value should not be trimmer. By default trim is false.
date_format	OPTIONAL	A string describing how the date should be formatted. By default it uses dd-MM-yyyy
position	REQUIRED(for DELIMITED)	For delimited files an integer explaining which field is to be used

Enum Mapping

An enum can be any default constructed enum class. Enums that have an explicit private constructor with arguments are not currently supported.

Attribute name	Required	Description
name	REQUIRED	the name of the attribute to use as the id
start	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the position this field starts at.
length	REQUIRED (for FIXED_LENGTH)	An integer whose value specifies the length of field
type	REQUIRED	The type of the attribute. Can be one of (string, long, short, date, integer)
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the

		DefaultPropertyAccessor is used.
trim	OPTIONAL	true if the type is string and the value should be trimmed (i.e preceding and proceeding spaces are removed. False if the value should not be trimmer. By default trim is false.
position	REQUIRED(for DELIMITED)	For delimited files an integer explaining which field is to be used

Object Mapping

An object can be any Canyon configured Java Bean.

Attribute name	Required	Description
name	REQUIRED	the name of the attribute
type	REQUIRED	The type of the attribute. This should be the full class name
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the DefaultPropertyAccessor is used.
join	REQUIRED	The field on the property class that should be used to join with the id field of this class

Collection Mapping

Canyon maps collections to the ArrayList class.

Attribute name	Required	Description
name	REQUIRED	the name of the collection attribute
type	REQUIRED	The type of the collection attribute. This should be the full class name of the classes the list will be holding.
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the DefaultPropertyAccessor is used.
join	REQUIRED	The field on the collection class that should be used to join with the id field of this class

Set Mapping

Canyon maps collections to the HashSet class.

Attribute name	Required	Description
name	REQUIRED	the name of the collection attribute
type	REQUIRED	The type of the set attribute. This should be the full class name of the classes the list will be holding.
accessor	OPTIONAL	A class that specifies how the value should be obtained. By default the DefaultPropertyAccessor is used.
join	REQUIRED	The field on the collection class that should be used to join with the id field of this class

Chapter 4 - Relationships and keys

In some cases you may have different files that are related to each other. Canyon gives the ability to model these relationships in a number of ways.

Relationship	Supported	Description
One to One	Yes	Use the <object> element in the mapping file
One to Many	Yes	Use the <collection> or <set> element in the mapping file
Many to One	No	No capability
Many to Many	No	No capability

Unlike a database Canyon only allows single field joins directly to another object. It does not map multi field or query type relationships. Canyon provides the capability to join a 'foreign' object with an entity based on the value of the id field of the entity matching the foreign field value. The foreign field is defined using the join attribute.

Also unlike a database Canyon does not scan and index the file. It relies upon the strict sequential ordering of data in the file as it iterates over the file. Remember even though in many ways Canyon provides similar mechanisms to the Hibernate ORM it is essentially iterating over a series of files.

The way this works is shown in Config 4

```
<!DOCTYPE mapping SYSTEM "http://canyon.hjcsolutions.co.uk/canyon_mapping.dtd">
<mapping>
  <class name="co.uk.hjc.canyon.example.Shop" type="DELIMITED" delimiter=",">
    <id name="id" position="0" type="integer"/>
    <property name="name" position="1" type="string" trim="true"/>
    <object name="manager" type="co.uk.hjc.canyon.example.Manager" join="shopId"/>
    <collection name="products" type="co.uk.hjc.canyon.example.Product"
      join="shopId"/>
    <set name="staff" type="co.uk.hjc.canyon.example.Employee" join="shopId"/>
  </class>
</mapping>
```

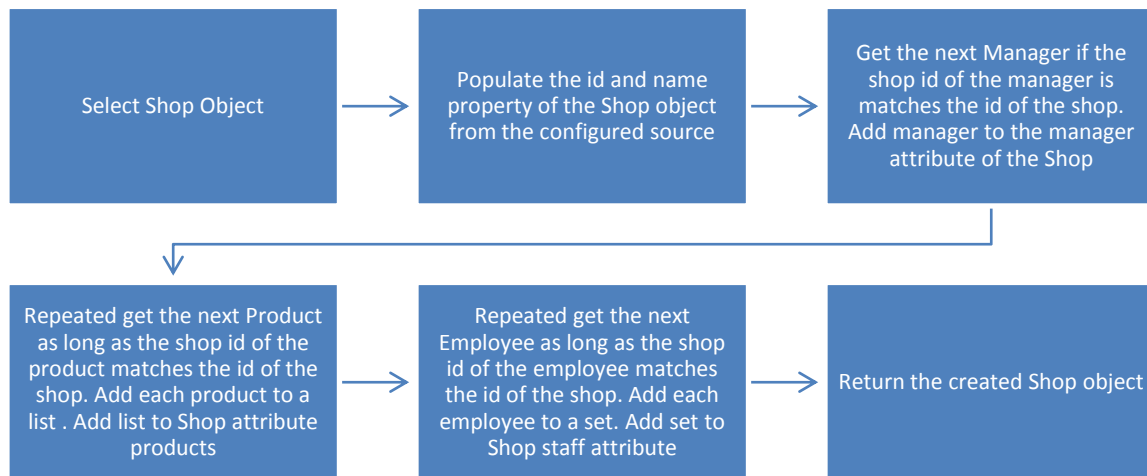
Config 4

The mapping description shows the way the three types of mapping can be used. In each case Canyon will use the value of the id field **id** of the Shop object to match the join field **shopId** in the Manager, Product and Employee object.

For relationships to operate the id field is mandatory, as this is integral to the mechanism used.

For this example the flow of execution is as shown in Figure 3. The diagram shows that Canyon essentially chains a sequence of calls together to create the resultant object.

Each piece of the chain iterates over the configured file. Therefore using the analogy of cursor, the cursor at the end of the flow will have been moved on for all four streams (Shop, Manager, Product, Employee). So the next call for any of these class would start from the position of the cursor at that point, not from the point of the last explicit call for that particular class.

**Figure 3**

It should be noted that the file formats of each class are completely independent from each other. Canyon provides the flexibility to manage different file formats for each class simultaneously.

We hope that you enjoy using the Canyon Framework. The src code and java docs so feel free to browse and modify for your own needs.