Language Guide for 2.0

MVEL has largely been inspired by Java syntax, but has some fundamental differences aimed at making it more efficient as an expression language, such as operators that directly support collection, array and string matching, as well as regular expressions. MVEL is used to evaluate expressions written using Java syntax.

In addition to the expression language, MVEL serves as a templating language for configuration and string construction. Another wiki page for a summary information is: https://en.wikipedia.org/wiki/MVEL.

MVEL 2.x expressions may consist of:

- Property expressions
- Boolean expressions
- Method invocations
- Variable assignments
- Function definitions

Basic Syntax

MVEL is an expression language based on Java-syntax, with some marked differences specific to MVEL. Unlike Java however, MVEL is dynamically typed (with optional typing), meaning type qualification is not required in the source.

MVEL interpreters as downloadable libraries which can be integrated to the product is available. It needs to be downloaded from the [https://maven.apache.org/ maven] website. The libraries expose APIs. If an expression it passed to the interface of the library, the expression is evaluated and result is provided.

An MVEL expression can be as simple as a single identifier, or as complicated as a full blown boolean expression with method calls and inline collection creations.

Simple Property Expression

```
user.name
```

In this expression, we simply have a single identifier (**user.name**), which by itself, is what we refer to in MVEL as a **property expression**, in that the only purpose of the expression is to extract a property out of a variable or context object. Property expressions are one of the most common uses, allowing MVEL to be used as a very high performance, easy to use, reflection-optimizer.

MVEL can even be used for evaluating a boolean expression:

```
user.name == 'John Doe'
```

Like Java, MVEL supports the full gambit of operator precedence rules, including the ability to use bracketing to control execution order.

```
(user.name == 'John Doe') &amp;amp;&amp;amp; ((x * 2) - 1) > 20
```

Multiple Statements

You may write scripts with an arbitrary number of statements using the semi-colon to denote the termination of a statement. This is required in all cases except in cases where there is only one statement, or for the last statement in a script.

```
statement1; statement2; statement3
```

Note the lack of semi-colon after statement3.

New lines are not substitutes for the use of the semi-colon in MVEL.

Returned Values

MVEL is designed to be an integration language at its core, allowing developers to provide simple scripting facilities for binding and logic. As such, MVEL expressions use a ‘’last value out” principle. This means, that although MVEL supports the `return` keyword, it is almost never needed. For example:

```
a = 10;
b = (a = a * 2) + 10;
a;
```

In this particular example, the expression returns the value of `a` as it is the last value of the expression. It is functionally identical to:

```
a = 10;
b = (a = a * 2) + 10;
return a;
```

Value Tests

All equality checks in MVEL are based on ‘’value’‘not’‘reference”. Therefore, the expression `foo == 'bar'` is the equivalent to `foo.equals("bar")` in Java.

Testing for Value Emptiness

MVEL provides a special literal for testing for emptiness of a value, cleverly named **empty**.

For example:

```
foo == empty
```

The example expression will be ‘’true” if the value of foo satisfies any of the requirements of emptiness.

Testing for Null

MVEL allows both the use of the keyword **null** or **nil** to represent a null value.

```
foo == null;
foo == nil; // same as null
```

Value Coercion

MVEL’s type coercion system is applied in cases where two incomparable types are presented by attempting to coerce the ‘’right’’value to that of the type of the’‘left” value, and then vice-versa.

For example:

```
"123" == 123;
```

This expression is ‘’true’‘in MVEL because the type coercion system will coerce the untyped number’‘123” to a String in order to perform the comparison.

Inline List, Maps and Arrays

MVEL allows you to express Lists, Maps and Arrays using simple elegant syntax. Consider the following example:

```
["Bob" : new Person("Bob"), "Michael" : new Person("Michael")]
```

This is functionally equivalent to the following code:

```
Map map = new HashMap();
map.put("Bob", new Person("Bob"));
map.put("Michael", new Person("Michael"));
```

This can be a very powerful way to express data structures inside of MVEL. You can use these constructs anywhere, even as a parameter to a method:

```
something.someMethod(["foo" : "bar"]);
```

Lists

Lists are expressed in the following format: ‘’[item1, item2, …]”

For example:

```
["Jim", "Bob", "Smith"]
```

Maps

Maps are expressed in the following format: ‘’[key1 : value1, key2: value2, …]”

For example:

```
["Foo" : "Bar", "Bar" : "Foo"]
```

Arrays

Arrays are expressed in the following format: ‘’{item1, item2, …}”

For example:

```
{"Jim", "Bob", "Smith"}
```

Array Coercion

One important facet about inline arrays to understand is their special ability to be coerced to other array types. When you declare an inline array, it is untyped, but say for example you are passing to a method that accepts int[]. You simply can write your code as the following:

```
foo.someMethod({1,2,3,4});
```

In this case, MVEL will see that the target method accepts an **int[]** and automatically type the array as such.

Property Navigation

MVEL property navigation follows well-established conventions found in other bean property expressions found in other languages such as Groovy, OGNL, EL, etc.

Unlike some other languages which require qualification depending on the underlying method of access, MVEL provides a single, unified syntax for accessing properties, static fields, maps, etc.

Bean Properties

Most java developers are familiar with and user the getter/setter paradigm in their Java objects in order to encapsulate property accessors. For example, you might access a property from an object as such:

```
user.getManager().getName();
```

In order to simplify this, you can access the same property using the following expression:

```
user.manager.name
```

> ***Note:*** *In situations where the field in the object is* ***public****, MVEL will still prefer to access the property via it's getter method.*

Null-Safe Bean Navigation

Sometimes you have property expressions which may contain a null element, requiring you to create a null-check. You can simplify this by using the null-safe operator:

```
user.?manager.name
```

This is functionally equivalent to writing:

```
if (user.manager != null) { return user.manager.name; } else { return null; }
```

Collections

Traversal of collections can also be achieved using abbreviated syntax.

List Access

Lists are accessed the same as array's. For example:

```
user[5]
```

is the equivalent of the Java code:

```
user.get(5);
```

Map Access

Maps are accessed in the same way as array's except any object can be passed as the index value. For example:

```
user["foobar"]
```

is the equivalent of the Java code:

```
user.get("foobar");
```

For Maps that use a String as a key, you may use another special syntax:

```
user.foobar
```

... Allowing you to treat the Map itself as a virtual object.

Strings as Arrays

For the purposes of using property indexes (as well as iteration) all Strings are treated as arrays. In MVEL you may refer to the first character in a String variable as such:

```
foo = "My String";
foo[0]; // returns 'M';
```

Literals

A literal is used to represent a fixed-value in the source of a particular script.

String literals

String literals may be denoted by single or double quotes.

```
"This is a string literal"
'This is also string literal'
```

String Escape Sequences

- `\\` - Double escape allows rendering of single backslash in string.
- `\n` - Newline
- `\r` - Return
- `\u####` - Unicode character (Example: `\uAE00`)
- `\###` - Octal character (Example: `\73`)

Numeric Literals

Integers can be represented in decimal (base 10), octal (base 8), or hexadecimal (base 16).

A decimal integer can be expressed as any number that does not start with zero.

```
125 // decimal
```

An octal representation of an integer is possible by prefixing the number with a zero, followed by digits ranging from 0 to 7.

```
0353 // octal
```

Hexidecimal is represented by prefixing the integer with `0x` followed by numbers ranging from 0-9..A-F.

```
0xAFF0 // hex
```

Floating Point Literals

A floating point number consists of a whole number and a factional part denoted by the point/period character, with an optional type suffix.

```
10.503 // a double
94.92d // a double
14.5f // a float
```

BigInteger and BigDecimal Literals

You can represent `BigInteger` and `BigDecimal` literals by using the suffixes `B` and `I` (uppercase is mandatory).

104.39484B // BigDecimal 8.4I // BigInteger ***

Boolean Literals

Boolean literals are represented by the reserved keywords `true` and `false`.

Null Literal

The null literal is denoted by the reserved keywords `null` or `nil`.

Type Literals

Type literals are treated pretty much the same as in Java, with the following format: ‘’<PackageName>.<ClassName>”.

So a class may be qualified as such:

```
java.util.HashMap
```

Or if the class has been imported either inline by or by external configuration—it is simply referred to by its unqualified name:

```
HashMap
```

Nested Classes

Nested classes are not accessible through the standard dot-notation (as in Java) in MVEL 2.0. Rather, you must qualify these classes with the `$` symbol.

```
org.proctor.Person$BodyPart
```

Flow Control

MVEL’s power goes beyond simple expressions. In fact, MVEL supports an assortment of control flow operators which will allow you to perform advanced scripting operations.

If-Then-Else

MVEL supports full, C/Java-style if-then-else blocks. For example:

```
if (var > 0) {
   System.out.println("Greater than zero!");
}
else if (var == -1) {
   System.out.println("Minus one!");
}
else {
   System.out.println("Something else!");
}
```

Ternary Statements

Ternary statements are supported just as in Java:

```
var > 0 ? "Yes" : "No";
```

And nested ternary statements:

```
var > 0 ? "Yes" : (var == -1 ? "Minus One!" : "No")
```

Foreach

One of the most powerful features in MVEL is it’s **foreach** operator. It is similar to the for each operator in Java 1.5 in both syntax and functionality. It accepts two parameters separated by a colon, the first is the local variable for the current element, and the second is the collection or array to be iterated.

For example:

```
count = 0;
foreach (name : people) {
   count++;
   System.out.println("Person #" + count + ":" + name);
}

System.out.println("Total people: " + count);
```

Since MVEL treats Strings as iterable objects you can iterate a String (character by character) with a foreach block:

```
str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

foreach (el : str) {
   System.out.print("["+ el + "]");
}
```

The above example outputs:

```
[A][B][C][D][E][F][G][H][I][J][K][L][M][N][O][P][Q][R][S][T][U][V][W][X][Y][Z]
```

You can also use MVEL to count up to an integer value (from 1):

```
foreach (x : 9) {
   System.out.print(x);
}
```

Which outputs:

```
123456789
```

Syntax Note: As of MVEL 2.0, it is now possible to simply abbreviate ‘‘foreach” in the same way it is in Java 5.0, by using the `for` keyword. For example:

```
for (item : collection) { ... }
```

For Loop

MVEL 2.0 implements standard C for-loops:

```
for (int i =0; i < 100; i++) {
   System.out.println(i);
}
```

Do While, Do Until

`do while` and `do until` are implemented in MVEL, following the same convention as Java, with `until` being the inverse of `while`.

```
do {
   x = something();
}
while (x != null);
```

… is semantically equivalent to …

```
do {
   x = something();
}
until (x == null);
```

While, Until

MVEL 2.0 implements standard `while`, with the addition of the inverse `until`.

```
while (isTrue()) {
   doSomething();
}
```

… or …

```
until (isFalse()) {
   doSomething();
}
```

Projections and Folds

Put simply, projections are a way of representing collections. Using a very simple syntax, you can inspect very complex object models inside collections.

Imagine you have a collection of **User** objects. Each of these objects has a **Parent**. Now say you want to get a list of all names of the parents (assuming the Parent class has a **name** field) in the hierarchy of users, you would write something like this:

```
parentNames = (parent.name in users);
```

You can even perform nested operations. Imagine instead, that the User object had a collection member called familyMembers, and we wanted a list of all the family members names:

```
familyMembers = (name in (familyMembers in users));
```

Assignments

MVEL allows you assign variable in your expression, either for extraction from the runtime, or for use inside the expression.

As MVEL is a dynamically typed language, you do not have to specify a type in order to declare a new variable. However, you may optionally do so.

```
str = "My String"; // valid
String str = "My String"; // valid
```

Unlike Java however, MVEL provides automatic type conversion (when possible) when assigning a value to a typed variable. For example:

```
String num = 1;
assert num instanceof String &amp;amp;&amp;amp; num == "1";
```

For dynamically typed variables where you simply want to perform a type conversion, you may simply cast the value to the type you desire:

```
num = (String) 1;
assert num instanceof String &amp;amp;&amp;amp; num == "1";
```

Function Definition

MVEL allows the definition of native functions with either the `def` or `function` keywords.

Functions are defined in-order of declaration, and cannot be foreword referenced. The only exception to this is ‘’within” functions themselves, it is possible to forward reference another function.

A Simple Example

‘’Defining a Simple Function”

```
def hello() { System.out.println("Hello!"); }
```

This defines a simple function called “hello” that accepts no parameters. When called it prints “Hello!” to the console. An MVEL-defined function works just like any regular method call, and resolution preference is to MVEL functions over base context methods.

```
hello(); // calls function
```

Accepting Parameters and Returning Values

Functions can be declared to accept parameters, and can return a single value. Consider the following example:

```
def addTwo(a, b) {
   a + b;
}
```

This function will accept two parameters (`a` and `b`) and then adds the two variables together. Since MVEL uses the ‘’last-value-out” principle, the resultant value will be returned. Therefore, you can use the function as follows:

```
val = addTwo(5, 2);
assert val == 10;
```

The `return` keyword can also be used to force a return a value from within the internal program flow of the function.

Closures

MVEL allows closures. However the functionality is not interoperable with native Java methods.

```
// define a function that accepts a parameter
def someFunction(f_ptr) { f_ptr(); }

// define a var
var a = 10;

// pass the function a closure
someFunction(def { a * 10 });
```

Lambda Expressions

MVEL allows the definition of lambda functions.

Example

‘’A simple lambda expression”

```
threshold = def (x) { x >= 10 ? x : 0 }; result = cost + threshold(lowerBound);
```

The above example defines a lambda and assigns it to the variable “threshold”. Lambda’s are essentially functions that are assignable to variables. They are essentially closures and can be used as such.

Interceptors

MVEL provides the ability to place interceptors within a ‘’compiled” expression. This can be particularly useful for implementing change listeners or firing external events based from within an expression.

An interceptor declaration uses the ‘’@Syntax”, similar to annotations in Java.

You declare interceptors before a statement you wish to enclose, and as such, an interceptor may implement either before or after listeners, or both. For example:

```
@Intercept
foreach (item : fooItems) {
   total += fooItems.price;
}
```

In this particular statement, the interceptor encloses the entire **foreach** block, so if the interceptor implements the after listener, that action will be fired when the foreach has completed.

The `org.mvel.integration.Interceptor` Interface

```
public interface Interceptor {
    public int doBefore(ASTNode node, VariableResolverFactory factory);
    public int doAfter(Object exitStackValue, ASTNode node, VariableResolverFactory factory);
}
```

The Interceptor interface provides two methods to be implemented: `doBefore` and `doAfter`. Here we will describe the meaning of the values passed into the methods from the MVEL runtime.

doBefore

The `doBefore` method is called before the enclosing statement is executed.

> “
>
> ***org.mvel.ASTNode::node***
>
> *The handle to the ASTNode is a reference to the ASTNode which is wrapped by the interceptor. This can be used to obtain information about the actual compiled code.*
>
> ***About AST API***
>
> *While access to the AST is provided, there is no guarantee that changes to the underlying AST API will not be changed in maintenance releases and subsequent revisions to MVEL, as certain bug fixes and performance improvements. Also certain compiler options may effect the structure of the AST and produce unexpected results, so understand this is an advanced feature.*
>
> ***org.mvel.integration.VariableResolverFactory::factory***
>
> *The variable resolver factory provides access to the current scope of variables in the expression. Please see 3Variable Resolvers for more information.*

doAfter

The `doAfter` method is called after the enclosing statement has executed.

> ***java.lang.Object::exitStackValue***
>
> *The `doAfter` method is executed after the enclosing statement has executed, but "not" before the end of the frame. Therefore, any value left on the stack at the end of the operation will still be present and therefore accessible to the interceptor. For example:*
>
> ```
> @Intercept cost += value;
> ```
>
> *In this particular case, the reduced value of "cost = cost + value" will be present on the stack until the end of the execution frame, and therefore this value will be available as the `exitStackValue` in the `doAfter` call.*
>
> ***org.mvel.ASTNode::node***
>
> *This is the same AST element as is passed to `doBefore`. See the last section for more details.*
>
> ***org.mvel.integration.VariableResolverFactory::factory***
>
> *See the last section for more details.*

Using Interceptors in the Compiler

Interceptors must be provided to the compiler prior to compiling the expression in order to wire the interceptors into the expression. So it should be noted here that interceptors ‘'may not" be used with the MVEL interpreter.

A Map is used to provide the compiler with the interceptors with the key representing the name of the interceptor, and the value being the interceptor instance.

For example:

```
// Create a new ParserContext
ParserContext context = new ParserContext();

Map<String, Interceptor> myInterceptors = new HashMap<String, Interceptor>();

// Create a simple interceptor.
Interceptor myInterceptor = new Interceptor() {
    public int doBefore(ASTNode node, VariableResolverFactory factory) {
        System.out.println("BEFORE!");
    }

    public int doAfter((Object value, ASTNode node, VariableResolverFactory factory) {
        System.out.println("AFTER!");
    }
};

// Now add the interceptor to the map.
myInterceptors.put("Foo", myInterceptor);

// Add the interceptors map to the parser context.
context.setInterceptors(myInterceptors);

// Compile the expression.
Serializable compiledExpression = MVEL.compileExpression(expression, context);
```

> ***Serializability of Expressions***
>
> *If you intend to serialize a compiled expression for re-use across multiple runtime instances, you must ensure that your interceptors are, themselves, Serializable otherwise you will encounter serialization problems since your Interceptor "instance" is added to the AST directly.*

Typing

MVEL is dynamically typed language, with static typing. Most users of MVEL will rely simply on dynamic typing. It's simple, and easy to work with.

```
a = 10; // declare a variable 'a'
b = 15; // declare a variable 'b';

a + b;
```

Dynamic Typing and Coercion

The most important aspect of a language like MVEL, which interacts directly with native Java objects, which themselves are statically typed, is that of ‘’type coercion‘‘. Since MVEL cannot truly treat say, a `java.lang.String` object as a `java.lang.Integer` object for math operations, it must be able to’‘coerce” one type to another.

Performance Considerations

This is of course a serious performance consideration when using something like MVEL as an integration point at a ‘’hot spot” of your application. For heavy processing load, coercion overload is mitigated by caches and the MVEL optimizers (available only for pre-compiled expressions). However it is not always possible to bypass the overhead of a hard coercion operation depending on what is being done.

However, if String variables are being injected into the runtime to be evaluated as Integers, it is simply not possible to prevent the runtime from needing to parse the strings into new Integer objects. This should always be considered.

Method Calls

Calling of methods is one of the most important aspects of coercion. Fundamentally, it allows you to directly call method without the need to finesse the inputs. Instead the interpreter or compiler will analyze the types being passed to the method, determine what coercion needs to be done to accomplish the call, and in the case of overloaded methods, choose the method which most closely matches the input types by avoiding using coercion if possible.

Arrays

Arrays are one of the more interesting aspects of the coercion system in MVEL, in that, MVEL uses untyped arrays by default (meaning `Object[]` arrays for all intents and purposes). Only when faced with an array type conflict, will MVEL attempt to ‘’coerce” the entire array to the needed input type. For example: in the case of a method call parameter.

Example:

```
myArray = {1,2,3};

// pass to method that accepts String[]
myObject.someMethod(myArray);
```

In this particular case, `someMethod()` accepts a `String[]` array. This will not fail in MVEL, instead MVEL will convert the array to a `String[]`.

Static Typing

Static typing in MVEL functions just as it does in Java, but by default still works in concert with coercion.

```
int num = 10;
```

This declares a typed integer variable called `num`. When you do this, the MVEL runtime **will** enforce the type. For example, trying to assign an incompatible type after the declaration will result in an exception:

```
num = new HashMap(); // will throw an incompatible typing exception.
```

However, MVEL ‘’will” perform coercion to a statically typed variable if the value being assigned is coercable.

```
num = "100"; // will work -- parses String to an integer.
```

> ***Only once in scope***
>
> *Once you have declared a statically typed variable, you cannot declare a new variable of the same name within the scope. For example:*

```
int i = 100;
int i = 200; // will throw an exception.
```

Strict Typing

Strict typing in MVEL is an optional mode for the compiler, in which all types must be fully qualified, either by declaration or inference.

Enabling Strict Mode

When compiling an expression, the compiler can be put into strict mode through the `ParserContext` by setting `setStrictTypeEnforcement(true)`.

Satisfying type strictness can be accomplished both by requiring explicit declaration of types inside the expression, or by notifying the parser ahead of time of what the types of certain inputs are.

For example:

```
ExpressionCompiler compiler = new ExpressionCompiler(expr);

ParserContext context = new ParserContext();
context.setStrictTypeEnforcement(true);

context.addInput("message", Message.class);
context.addInput("person", Person.class);

compiler.compile(context);
```

In this example we inform the compiler that this expression will be accepting two external inputs: `message` and `person` and what the types of those inputs are. This allows the compiler to determine if a particular call is safe at compile time, instead of failing at runtime.

Shell

The MVEL interactive shell allows you to directly interface with MVEL, and explore features of the MVEL language.

See also https://en.wikipedia.org/wiki/Drools https://en.wikipedia.org/wiki/Unified_Expression_Language

Launching the Shell

To launch the MVEL shell, simply run the MVEL distribution JAR as an executable jar:

```
java -jar mvel.jar
```

Alternatively, you can run the shell from within your favorite IDE by setting up a run profile for the class: `org.mvel2.sh.Main`.

Language FAQ

Why doesn't the .class reference work?

MVEL does not have a special ''.class" identifier to refer to type literals like Java. There are no class literals per se. Instead you refer to a class reference simply by its name. For example, if a method accepts type `Class` as a parameter you would call it just like this:

```
// MVEL
someMethod(String);

// Java-equivalent
someMethod(String.class);
```

In fact, MVEL treats ''.class" as a regular bean property. So by writing `String.class` the value returned will be an instance of `java.lang.Class` referring to `java.lang.Class` itself, since it would be the equivalent of writing `String.class.getClass()` in Java.

The principle reason for this, is that MVEL has a dynamic type system which treats types as regular variables, rather than qualified type-literals like in Java. And as such, MVEL allows for class types to be referenced as ordinary variables unlike Java, allowing for type-aliasing.

Why can't I write object.class.name?

This is a limitation which may be addressed in a future version of MVEL, but bean properties are ''not" supported against `Class` references. This does not mean that you cannot call methods of `Class`, but that you must use full qualified method calls like:

```
someVar.class.getName();     // Yes!
someVar.class.name;          // No!

someVar.getClass().getName() // Yes!
someVar.getClass().name      // No!
```

This limitation is completely limited to `java.lang.Class` as a consequence of property-resolution orders and the way that MVEL deals with class references.

MVEL 2.0 Templating Guide

MVEL 2.0 offers a new, more powerful, and unified templating engine to bring together many of the template concepts that were introduced in 1.2. Unfortunately, the architecture of the template engine in 1.2 was inadequate for regular maintenance, and the decision to completely re-write the template engine from the ground up was made.

MVEL 2.0 Basic Templating

MVEL Templates are comprised of *orb-tags* inside a plaintext document. Orb-tags denote dynamic elements of the template which the engine will evaluate at runtime.

If you are familiar with FreeMarker, this type of syntax will not be completely alien to you.

A Simple Template

```
Hello, @{person.getSex() == 'F' ? 'Ms.' : 'Mr.'} @{person.name}

This e-mail is to thank you for your interest in MVEL Templates 2.0.
```

This template shows a simple template with a simple embedded expression. When evaluated the output might look something like this:

```
Hello, Ms. Sarah Peterson

This e-mail is to thank you for your interest in MVEL Templates 2.0.
```

Escaping the @ Symbol

Naturally, since the @ symbol is used to denote the beginning of an orb-tag, you may need to *escape* it, to prevent it from being processed by the compiler. Thankfully, there is only one situation where this is necessary: when you actually need to produce the string '@{' as output in your template.

Since the compiler requires a combination of @ and { to trigger the orb recognition, you *can* freely use @ symbols without escaping them. For example:

```
Email any questions to: foo@bar.com

@{date}
@include{'disclaimer.html'}
```

But in the case where you need an @ symbol up-against an orb-tag, you will need to escape it by repeating it twice:

```
@{username}@@@{domain}
```

That's two @'s to escape one symbol, and the third @ being the beginning of the tag. If this looks too messy, you can always use the alternate approach of using an expression tag, like this:

```
@{username}@{'@'}@{domain}
```

MVEL 2.0 Orb Tags

This page contains a list of all *orb-tags* available out-of-the-box in the MVEL 2.0 templating engine.

@{} *Expression Orb*

The expression orb is the most rudimentary form of orb-tag. It contains a value expression which will be evaluated to a string, and appended to the output template. For example:

```
Hello, my name is @{person.name}
```

@code{} *Silent Code Tag*

The silent code tag allows you to execute MVEL expression code in your template. It does not return a value and does not affect the formatting of the template in any way.

```
@code{age = 23; name = 'John Doe'}
@{name} is @{age} years old.
```

This template will evaluate to: *John Doe is 23 years old*.

@if{}@else{} *Control Flow Tags*

The **@if{}** and **@else{}** tags provide full if-then-else functionality in MVEL Templates. For example:

```
@if{foo != bar}
   Foo not a bar!
@else{bar != cat}
   Bar is not a cat!
@else{}
   Foo may be a Bar or a Cat!
@end{}
```

All blocks in MVEL Templates *must* be terminated with an @end{} orb, except in cases of an if-then-else structure, where @else{} tags denote the termination of the previous control statement.

@foreach{} *Foreach iteration*

The foreach tag allows you to iterate either collections or arrays in your template. Note: that the syntax for foreach has changed in MVEL Templates 2.0 to standardize the foreach notation with that of the MVEL language itself.

```
@foreach{item : products}
 - @{item.serialNumber}
@end{}
```

MVEL 2.0 *requires* you specify an iteration variable. While MVEL 1.2 assumed the name *item* if you did not specify an alias, this has been dropped due to some complaints about that default action.

Multi-iteration

You can iterate more than one collection in a single foreach loop at one time by comma-separating the iterations:

```
@foreach{var1 : set1, var2 : set2}
  @{var1}-@{var2}
@end{}
```

Delimiting

You can automatically add a text delimiter to an iteration by specifying the iterator in **@end{}** tag.

```
@foreach{item : people}@{item.name}@end{', '}
```

This would return something like: *John, Mary, Joseph*.

@include{} *Include Template File*

You may include a template file into an MVEL template using this tag.

```
@include{'header.mv'}

This is a test template.
```

You may also execute an MVEL expression inside an include tag by adding a semicolon after the template name:

```
@include{'header.mv'; title='Foo Title'}
```

@includeNamed{} *Include a Named Template*

Named templates are templates that have been precompiled and passed to the runtime via a TemplateRegistry, or templates that have been declared within the template itself. You simply include:

```
@includeNamed{'fooTemplate'}
@includeNamed{'footerTemplate', showSomething=true}
```

You may also execute MVEL code in an @includeNamed{} tag, just as with the @include{} tag.

@declare{} *Declare a Template*

In addition to including external templates from external files, and passing them in programmatically, you can declare a template from within a template. Which allows you to do things like this:

```
@declare{'personTemplate'}
 Name: @{name}
 Age:  @{age}
@end{}

@includeNamed{'personTemplate'; name='John Doe'; age=22}
```

@comment{} *Comment tag*

The comment tag allows you add an invisible comment to the template. For example:

```
@comment{
  This is a comment
}
Hello: @{name}!
```

MVEL 2.0 Template Integration

Using MVEL templates is straight-forward and easy. Like regular MVEL expressions, they can be executed interpretively, or be pre-compiled and be re-used for faster evaluation.

The `org.mvel.templates.TemplateRuntime` Class

The `TemplateRuntime` class is the center of the template engine. You can pass a template to be evaluated to the template engine by way of the `eval()` method.

In general, the template engine follows all the same rules for context and variable binding, with an overloaded set of `eval()` methods.

Here's a simple example of parsing a template interpretively:

```
String template = "Hello, my name is @{name.toUpperCase()}");
Map vars = new HashMap();
vars.put("name", "Michael");

String output = (String) TemplateRuntime.eval(template, vars);
```

At the end of execution, the "output" variable will contain the string:

```
Hello, my name is MICHAEL
```

The `org.mvel.templates.TemplateCompiler` Class

The `TemplateCompiler` class allows for pre-compilation of the templates.

When you compile a template, a compact, reusable evaluation tree is produced that can be quickly used to evaluate a template. It is used straightforwardly:

```
String template = "1 + 1 = @{1+1}";

// compile the template
CompiledTemplate compiled = TemplateCompiler.compileTemplate(template);

// execute the template
String output = (String) TemplateRuntime.execute(compiled);
```

At the end of execution, the "output" variable will contain the string:

```
1 + 1 = 2
```