

Living Systems® Process Suite

Expression Language

Living Systems Process Suite Documentation

3.2
Tue Jan 12 2021

Whitestein Technologies AG | Hinterbergstrasse 20 | CH-6330 Cham
Tel +41 44-256-5000 | Fax +41 44-256-5001 | <http://www.whitestein.com>

Copyright © 2007-2021 Whitestein Technologies AG
All rights reserved.

Copyright © 2007-2021 Whitestein Technologies AG.

This document is part of the Living Systems® Process Suite product, and its use is governed by the corresponding license agreement. All rights reserved.

Whitestein Technologies, Living Systems, and the corresponding logos are registered trademarks of Whitestein Technologies AG. Java and all Java-based trademarks are trademarks of Oracle and/or its affiliates. Other company, product, or service names may be trademarks or service marks of their respective holders.

Contents

1	Expression Language	1
2	Expressions	3
2.1	Chaining Expressions	3
2.2	Block	3
2.3	Literals	4
3	Data Types	5
3.1	Casting	6
3.2	Object	6
3.3	void	6
3.4	Simple Data Types	6
3.4.1	Binary	6
3.4.2	String	7
3.4.3	Boolean	7
3.4.4	Integer	8
3.4.5	Decimal	8
3.4.6	Date	9
3.5	Complex Data Types	10
3.5.1	Collections	10
3.5.1.1	List	10
3.5.1.2	Set	11
3.5.2	Map	11
3.5.3	Reference	12
3.5.4	Closure	13
3.5.5	User-Defined Record	14
3.5.5.1	Methods	14
3.5.5.2	Constructors	17
3.5.6	Property Path	19
3.5.7	Property	20
3.5.8	Type	20
3.5.9	Enumeration	20
3.5.10	Null	21

4	Operators	23
4.1	Assignment Operator	23
4.2	Arithmetic Operators	23
4.3	Compound-Assignment Operators	24
4.4	Logical Operators	24
4.5	Comparison Operators	24
4.6	Concatenation	25
4.7	Reference and Dereference Operators	26
4.8	Inclusion Operator	26
4.9	Namespace Operator	27
4.10	Selector Operator	27
4.11	Ternary Conditional	27
4.12	Null-Coalescing Operator	28
4.13	Access Operator	28
4.14	Safe-Dot Operator	28
4.15	Cast Operator	29
4.16	Evaluation Order	29
5	Comments	31
6	Local Variables	33
7	Controlling Flow	35
7.1	Branching	35
7.1.1	if-then-end	35
7.1.2	if-then-else-end	35
7.1.3	if-then-elsif-end	36
7.1.4	if-then-elsif-then-else-end	36
7.1.5	switch	36
7.2	Looping	37
7.2.1	for	37
7.2.2	foreach	37
7.2.3	while	38
7.2.4	break	38
7.2.5	continue	38

8	Exception Handling	39
8.1	Throwing Exceptions	39
8.2	Catching Exceptions	40
8.3	Built-in Errors	40
9	Functions	41
9.1	Functions Implemented by a Java Method	42
9.2	Function Annotations	42
9.3	Calling Functions	42
10	Model Elements	45
11	Reserved Words	47

Chapter 1

Expression Language

The *LSPS Expression Language* is a language used in models of the Living Systems® Process Suite to define element properties, definition implementations, assignments, etc. As such, it provides support for object-oriented and functional programming.

Note that it is not a full-fledged programming language and it is not possible to define models exclusively in the LSPS Expression Language; the language is adapted to support your modeling efforts.

For example, you design a BPMN process with a Log task, you define the log message as an expression or expression block that results in a String in the Log property:

```
"Process " + processID + " started."
```

This expression uses String literals "Process " and " started", and the variable processID. On runtime, the system fetches the value of processID and concatenates the strings.

However, you cannot design the process flow with the Start Event and Log task in the Expression Language; For this purpose, you will use constructs of the [GO-BPMN Modelling Language](#).

Chapter 2

Expressions

An expression is a combination of literals, data types, keywords, variables, function calls, operators, and calls to model elements. It always returns its resulting value. For example, `"Hello " + getWorld()` concatenates the strings, `"Hello "` and the string returned by the `getWorld()` function call, and returns the concatenation.

```
//returns the right side "hello":
def String var := "hello";
```

2.1 Chaining Expressions

To chain multiple expression into a single expression, connect them with a semi-colon or a new line. Such a chained expression returns the return value of the last expression in the chain. Intermediary return values of the other chained expressions are ignored.

```
def String varString; varString := "Hello" //two expressions chained by a semi-colon (;)
varString:=varString + "World" //an expression chained by a new line
```

2.2 Block

An expression block is an expression with its own contexts.

Expression blocks follow the visibility rules of contexts: the data in an expression block, such as, an expression variable, cannot be access from outside of the block.

The `if`, `then`, `else`, `switch`, `foreach`, `while`, `for` content represent an expression block. You can create an expression block explicitly as well: start the block with the `begin` keyword and finish it with the `end` keyword.

```
begin
//declaration of an expression variable:
def Boolean visibility := true
  //beginning a codeblock
  begin
    //declaration of an expression variable in the block (not available out of the block):
    def Boolean visibility := false;
    visibility == false;
  end;
visibility == true;
end;
```

2.3 Literals

Literals represent fixed values. The notation of literals depends on the data type they represent. The notation is documented in [Data Types](#).

Chapter 3

Data Types

The *LSPS Expression Language* is a statically typed language: That means that all values are of a given data type which does not change on runtime: If you create a String literal it cannot spontaneously act as an Integer; When you declare a local string variable, it can only reference a string value, etc.

Data types are part of the data type hierarchy with the [Object](#) data type as the root of the tree with the sole exception being the [void](#) type. Depending on their position in the hierarchy, they inherit properties from each other: one type becomes the supertype of another type. An object of a data type can be used anywhere where you can use its super type. For example, you can assign a variable of type Decimal also a value of type Integer, since Integer is a subtype of Decimal. These relationships apply to built-in as well as [custom data types](#) in a hierarchy.

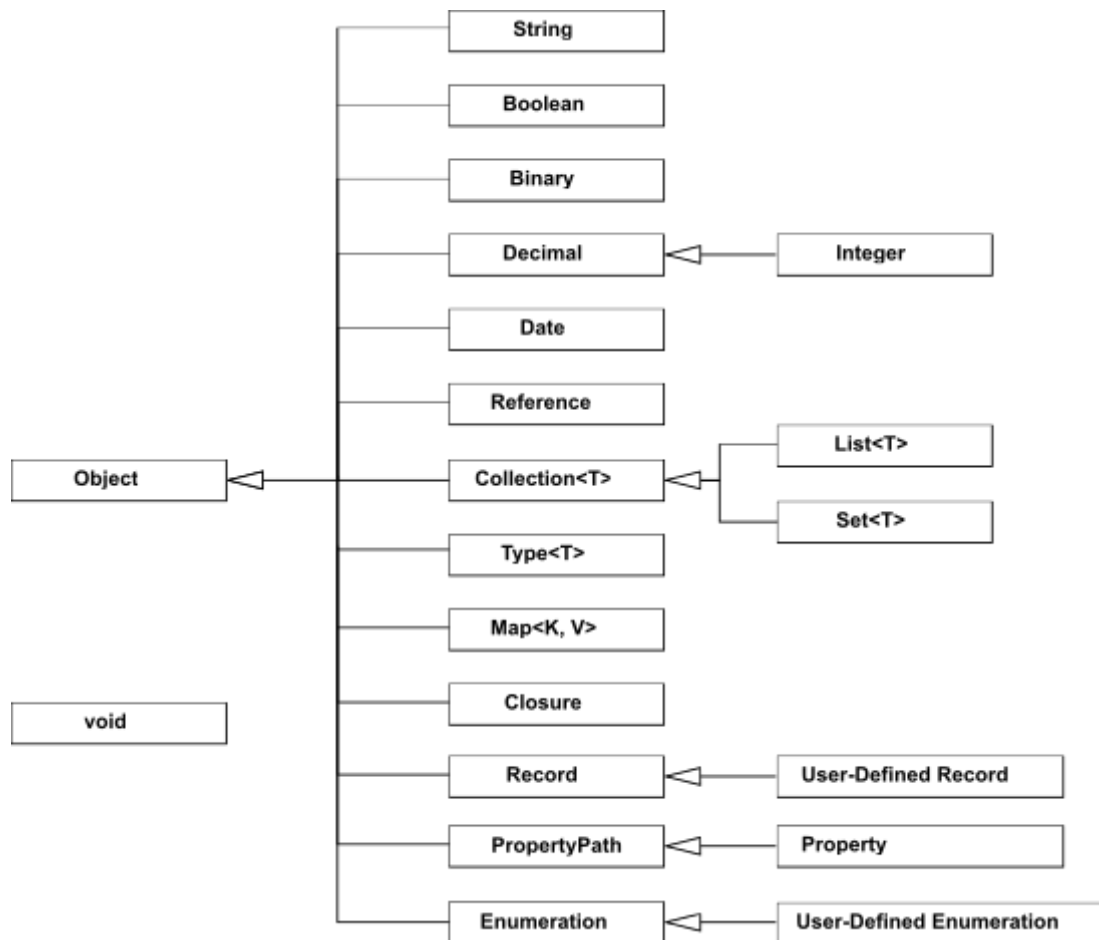


Figure 3.1 Type hierarchy of built-in types

3.1 Casting

Casting takes place when you change the type of a value to another type. When widening a type, that is changing a value of a subtype to its supertype, the type change occurs automatically. When narrowing-down a type, you need to cast the type explicitly, for example:

```
person as NaturalPerson
```

You can check the type of an object with the `instanceof` operator.

Note: Alternatively, you can use the `cast()` function of the Standard Library, for example, `cast(o, type(D))`

3.2 Object

The *Object* data type is the super type of all data types and therefore the only data type without a supertype. It is represented by *java.lang.Object*. Therefore if you want to allow any data type, for example, as a parameter, use the *Object* data type. It is represented by the *java.lang.Object* class. All objects can hold the `null` value.

3.3 void

The *void* type is used as the return value of `methods`, `functions`, and `closures` that do not return any value.

3.4 Simple Data Types

Simple data types contain values without further data type structuring.

3.4.1 Binary

Objects of binary data type are typically used to hold binary data when downloading and uploading files, or working over binary database data.

It is represented by *com.whitestein.lsp.lang.exec.BinaryHolder*.

Binary literals are not supported.

To define the type:

```
Binary
```

3.4.2 String

A **String** holds a sequence of Unicode characters. The data type is implemented by the `java.lang.String` class.

To define the type:

```
String
```

To create an instance:

```
"Sequence of Unicode characters"
```

A String can contain special characters defined using their ASCII codes. For example, you can use the ASCII tab code (#9) to have a tab in your String, the line feed character (#10) to insert the end of a line and make a multi-line String, etc., for example:

```
//newline character is #10
"This" + #10 + "is" + #10 + "a" + #10 + "multiline" + #10 + "string with "+
"multiple" + #10 + " new lines " + #10 + " which represent " + #10 + "line breaks."
```

To escape characters, use the double-quote (") character:

```
"This is all one string: ""Welcome to String escaping!"""
```

To annotate a string that is not to be localized, add the hashtag # sign in front of the string.

```
#"This is a string value which will not be localized."
```

The hashtag # sign signals that the String is to remain unlocalized and that this was the intention of the developer. Such Strings are **excluded from checks of unlocalized Strings**.

To create a local variable:

```
def String s := "My String"
```

3.4.3 Boolean

Boolean objects hold the values `null`, `true` or `false`. It is implemented by the `java.lang.Boolean` class.

To define the type:

```
Boolean
```

To create an instance:

```
true
```

To create a local variable:

```
def Boolean s := true
```

The expression `def Boolean boolVar := true` declares and defines a local variable of type.

3.4.4 Integer

Integer objects hold natural numbers and their negatives. The data type is a subtype of the *Decimal* type. It is represented by *com.whitestein.Isps.lang.Decimal*.

Note that the underlying Java type is *BigDecimal*; hence no relevant maximum limit applies to the value.

To define the type:

```
Integer
```

To create an instance:

```
-100
```

To create a local variable:

```
def Integer i := 42
```

3.4.5 Decimal

Decimal objects hold numerical fixed-point values. It is represented by *com.whitestein.Isps.lang.Decimal*.

Note: A *Decimal* type is internally represented by two integer values: an unscaled value and a scale. The value is hence given as $\langle \text{UNSCALED_VALUE} \rangle * 10^{**} \langle \text{SCALE_VALUE} \rangle$. The integer scale defines where the decimal point is placed on the unscaled value. The scale is a 32-bit integer. If zero or positive, the scale is the number of digits to the right of the decimal point. If negative, the unscaled value is multiplied by ten to the power of the negation of the scale. Decimal values are, for example, $2e+12$, $-1.2342e0$, 1.0 .

When assigning a value of type *Decimal*, you need to define the scale and rounding mode. The following rounding modes can be used on decimals:

- **UP**: rounds away from zero
 - **DOWN**: rounds towards zero
 - **CEILING**: rounds towards positive infinity
 - **FLOOR**: rounds towards negative infinity
 - **HALF_UP**: rounds towards the nearest neighbor unless both neighbors are equidistant, in which case it rounds up
 - **HALF_DOWN**: rounds towards nearest neighbor unless both neighbors are equidistant, in which case it rounds down
 - **HALF_EVEN**: rounds towards the nearest neighbor unless both neighbors are equidistant, in which case, it rounds towards the even neighbor
 - **UNNECESSARY**: asserts that no rounding is necessary
-

The rounding mode is defined for decimal variables or for a record field of type `Decimal`.

To define the type:

```
Decimal
```

To define a `Decimal` type with Scale 100 and Rounding Model UP:

```
Decimal(100, UP)
```

To create an instance:

```
-10.0;  
7E-3 //0.007  
6.63E+34
```

To create a local variable:

```
def Decimal intVar := 100
```

Important: Decimal values are normalized if they contain 0 digits after the decimal point: decimal value 1.0 and integer value 1 have the same numerical value and therefore `1.0 == 1`.

3.4.6 Date

The **Date** object holds a specific time. It is represented by *java.util.Date*.

Important: Only dates since the introduction of the Gregorian calendar, that is, the year 1582 are supported: for Dates that occurred before, a shift in days can occur rendering the date incorrect.

To define the type:

```
Date
```

To create an instance:

```
d'yyyy-MM-dd HH:mm:ss.SSS'
```

To create a local variable:

```
def Date myDate := d'2015-12-24 20:00:00.000'
```

Important: When constructing a date, consider using the date functions from the Standard Library.

3.5 Complex Data Types

Complex data types are composite data types based on other data types.

3.5.1 Collections

A *Collection* is a groupings of items of a particular data type. It is represented by *com.whitestein.lsp.lang.exec.CollectionHolder*.

Collections are ordered and immutable: once a collection is created, it cannot be changed, while you can change individual collection items. Each item of the collection represents an expression

To access an element of a Collection, you need to specify the position of the element in the List. Note that the first element of a List is on position zero. For example, `[10, 20, 30][1]` returns 20 as position 0 of this list points to the value 10, the first element of the list.

Hierarchy of collections follows the hierarchy of their elements: `List<TA>` is a subtype of `List<TB>`, if `TA` is a subtype of `TB`. `Set<TA>` is a subtype of `Set<TB>`, if `TA` is a subtype of `TB`.

To define the type:

```
Collection<T>
```

3.5.1.1 List

A **List** represents an ordered collection of items of a type with possible duplicate values. It is represented by the *com.whitestein.lsp.lang.exec.ListHolder* class.

To define the type:

```
List<T>
```

To create an instance:

```
["a", "b", "c", "d"]
```

List of lists

```
[["a", "b"], ["b", "c", "a"]]
```

To access an item in a List: Specify the position of the item in the List starting from 0: For example, `[10, 20, 30][1]` returns 20 as position 0 points to the value 10, the first element.

Unlike on Sets, accessing list items is performance-wise efficient.

3.5.1.1.1 Creating a List of Integers with the Range Operator

To create a List of Integers, you can use the `..` operator, the range operator:

```
1..5
//is equivalent to [1, 2, 3, 4, 5]
~
~
3..1
//is equivalent to [3, 2, 1]
```

Note that `toString(x..y)` returns `"x..y"`.

3.5.1.2 Set

A **Set** represents an ordered collection of items of a type with no duplicate values. It is represented by the `com.whitestein.lsp.lang.exec.SetHolder` class.

To define the type:

```
Set<T>
```

To create an instance:

```
//Set<Integer>:
{1,2,3};
```

To access an item in a Set: Specify the position of the item in the Set starting from 0: For example, `{10, 20, 30}[1]` returns 20 as position 0 points to the value 10, the first item.

Unlike in Lists, accessing items is performance-wise inefficient.

3.5.2 Map

Maps hold keys with their values and are immutable: once a map is created, it cannot be changed. You can change its key-value pairs, however, the map itself cannot be modified.

It is represented by `com.whitestein.lsp.lang.exec.MapHolder`.

To define the type:

```
//type map with the K type of its keys and the V type of its values:
Map<K, V>
```

Note that hierarchy of maps follows the hierarchy of its key and value types: `Map<KA, VA>` is a subtype of `Map<KB, VB>`, if `KA` is a subtype of `KB` and `VA` is a subtype of `VB`.

To create an instance:

```
[1->"a", 2->"b"]
```

To initialize an empty map, use the empty-map operator `[->]`:

```
def Map<Object, Object> myEmptyMap := [->];
```

To get a value of a key, specify the key for the appropriate value in square brackets, for example, `["firstKey"->"a", "anotherKey"->"b"] ["anotherKey"]` returns the string "b".

3.5.3 Reference

A **Reference** holds a reference expression that resolves to a variable or a record field (slot), not their value.

A Reference is conceptually similar to pointers in other languages. However, a Reference points to a variable or a record field, not to memory slot.

- A Reference to a variable resolves to the variable object
- A Reference to a record field resolves to the record instance and the association path.

The data type is represented by *com.whitestein.lsp.lang.exec.ReferenceHolder*.

To define the type:

```
Reference<T>
```

To create an instance:

```
&<TARGET>
```

To get the value in a Reference:

```
*variableWithReference
```

For more information, see [Reference and Dereference Operators](#).

Example use of referencing

```
//creates new Patient record instance with the diagnosis "flu":
def Patient r := new Patient(diagnosis -> "flu");
~
//creates the local variable status that holds the reference to the diagnosis field of the Patient
def Reference<String> status := &(r.diagnosis);
~
//function sets the status to cured on the patient:
setStatusToCured(&r.status);
~
//implementation of the setStatusToCured() function:
//def Reference<String> status:= statusReference;
//*status:="cured";
```

3.5.4 Closure

A closure is an anonymous function that can declare and make use of local variables in its own namespace and use variables from its parent expression context as opposed to lambdas. It is represented by *com.whitestein.lsp*s.↔ *lang.exec.ClosureHolder*.

To define the type:

```
//Syntax: { <INPUT_TYPE_1>, <INPUT_TYPE_2> : <OUTPUT_TYPE> }
{ String : Integer}
//Closure that has no parameters and returns an Object:
{ : Object}

def { String : String } myClosureVar := { s:String-> callingFunction(s) }
myClosureVar("my string")
```

Subtyping in closures is governed by their parameters and return type: Closure A is a subtype of closure B when:

- the return type of closure A is a subtype of the return type of closure B
- and parameter types of closure B are subtypes of parameter types of closure A.

{ S1,S2,... : S } is subtype of { T1, T2,... : T } when T1 is subtype of S1, T2 is subtype of S2, etc. and S is subtype of T.

To create an instance:

```
//Syntax: { <PARAMETERS> -> <CLOSURE_BODY> }
{s:String -> length(s)}
```

Parameter types can be omitted:

```
{s -> length(s)}
```

The system attempts to infer the type of closure arguments and its return value. Note that the types might be resolved incorrectly. To prevent such an event, consider defining the argument type explicitly as in the example.

To evaluate a closure use the parentheses () operator with the closure arguments:

```
def {Integer:String} closureVar := {x:Integer -> toString(x)};
def String closureResult := closureVar(3);
```

Important: It is recommended to always explicitly define the type of the closure input arguments, in the example, we explicitly define that *x* is an *Integer* in *x : Integer*.

3.5.5 User-Defined Record

A user-defined **Record** is the subtype of the Record type. It serves to create custom structured data types. A Record can define:

- **relationships** with another Records
- **methods** declared in a dedicated methods file

It is not possible to declare a Record type in the Expression Language: Records are modeled in the **data-type editor** of the Living Systems® Process Design Suite. However, you can create instances of Records. When you pass a Record, for example, as a function argument, it is passed by reference.

It is represented by *com.whitestein.lsp.lang.exec.RecordHolder*.

To create an instance:

```
new <RECORD> (<NAME_OF_FIELD> -> <FIELD_VALUE>, <NAME_OF_FIELD> -> <FIELD_VALUE>, ...)
```

For example:

```
new Book(genre -> Genre.FICTION, title -> "Catch 22", author -> new Author(name -> "Heller, Josep"))
```

Use the **dot operator** to access Record Fields or Properties (Fields of related Records using the Relationship name), and methods.

Example: *def* declares a new variable of the MyRecord type. *new* creates a new MyRecord instance, which is assigned to the variable (the instance is the proper memory space with the record) . Variable *r* points to the MyRecord instance and it is returned by the expression.

```
new <RECORD_NAME> (<PROPERTY> -> <PROPERTY_VALUE>)
```

3.5.5.1 Methods

A *method* is a piece of code similar to function but defined for a specific Record that performs an action on the Record instance. The action is defined as the method body: Method body can use Record's fields, methods, global variables and global functions. If the Record has a supertype, also the public methods of the supertype Record are accessible and can be overridden by the subtype method. To call the methods of the supertype Record, use the **super keyword**.

Important: While you can override the *toString()* method of Records so that when you call the *toString()* method on your Record, the system will use your method; however, it is not possible to override the *equals()* and *hashCode()* methods.

Methods are **declared in method definition files** with every Record having its own dedicated methods file.

Methods define their **visibility**:

- *private*: accessible only for the Record and its methods (define methods to access from outside)
- *protected*: accessible only from within the data type hierarchy (can be used by any subtype)
- *public*: accessible from anywhere

Methods support overloading, that is, a Record can define multiple methods with the same name as long as they have different arguments.

3.5.5.1.1 Declaring Methods

When declaring methods and constructors, you can reference the current Record object with the 'this' keyword

Methods of a Record are declared and defined for the given Record in the method definition:

```
<RECORD_NAME> {
~
    <METHOD_DEFINITIONS>
~
}
```

Individual methods are declared with the following syntax:

```
/** <description> */
<visibility> <flag> <returnType> <name> (<arguments>){
    <methodImplementation>
}
```

Note: To indicate that a method does not return any value, set the return type to `void`. It is not recommended to use `Null`.

Example method:

```
public String getName(){
    this.title;
}
```

When declaring methods, you can use the `this` keyword to reference the current Record instance:

```
this.callingRecordMethod();
```

Note that calling method fails with an exception if the object is `null` (safe-dot operator equivalent is not available).

3.5.5.1.2 Accessing Methods of Super Records

To access methods of Record's supertype, use the `super` keyword:

```
//supertype default constructor:
super();
~
super.methodFromParent();
```

You can override the inherited methods.

3.5.5.1.3 Extension Methods

Extension methods are special function types that are defined for a particular data type. Unlike common methods, they enable you to add methods to existing types, and that, including types from Standard Library, so you do not need to create a new type for your method.

Extension methods are declared the type they are defined for in a function definition file, which cannot be created with the Expression Language. They can be called either [like methods](#) or as functions.

Important: Extension methods are resolved as functions (not as methods) so that the extension method in the lowest context is used.

To declare an extension method on an existing Type, use the following syntax:

```
@ExtensionMethod
<visibility> <returnType> <nameOfExtensionMethod>(<extendedType><inputParams>) {
    <implementation>
}
```

Example implementation:

```
@ExtensionMethod
private String extensionMethodOfString(Integer i) {
    "String value returned by the extensionMethodOfString() over Integer " + i
}
```

Example calls of an extension method:

```
EntityGrid.removeAllColumns();
//or
removeAllColumns(EntityGrid);
```

3.5.5.1.4 Calling Methods

Record types can define methods that can be called on their Record instances or their Record, in the case of static methods:

```
<record_instance>.<method>(<parameters>)
```

To operate directly over the return values of methods, you can chain the methods:

```
<recordInstance>.<method1>.<method2>;
```

Note: Methods cannot use named parameters as its arguments: calls like `<record_instance>.<method>(<parameter> -> <value>)` is not supported).

3.5.5.1.5 Declaring Static Methods

You can declare a method as a *static* method: static methods are methods that belong to the Record, not a Record instance and as such are shared between all instances of the Record.

Important: Static methods can be declared only on Records. It is not possible to create static methods on Interfaces.

To declare a method as static, you need to prepone the method with the keyword *static*.

3.5.5.1.6 Abstract Records and Methods

If a Record is abstract, it cannot be instantiated, but it can be used as a super Record. An abstract Record can define abstract methods: abstract methods do not contain any implementation but any child Records must implement abstract methods of their super Records.

Abstract Records and methods serve to ensure that their child Records inherit the Fields and methods: the child Record inherits the Fields and must implement the abstract methods of the abstract Record.

When depicted in Diagrams, names of abstract Records and Methods are in italics.

An abstract method is declared as

```
public abstract Integer getBusinessData();
```

Note: Records are marked as *abstract* using the abstract flag, which is a Modeling Language feature: it is not possible to mark a Record as abstract only using the Expression Language.

3.5.5.2 Constructors

Constructors are Record methods that create and return an instance of the Record.

3.5.5.2.1 Constructor Availability

- If you do not declare a constructor, the following constructors are available:
 - If neither your Record nor its super Records declare a constructor, you can use the **field constructor**:

```
new Record(property1 -> Value1, property2 -> value2, ...);
```

//field constructor with no arguments:

```
new Record();
```

Note that you can assign a property value only to the visible properties (properties that are public or protected if visible from the given location).
 - If one of the super Records declares a non-parametric constructor and it can be reached from the child records using the `super()` constructor, you can use the default constructor:

```
new Record();
```

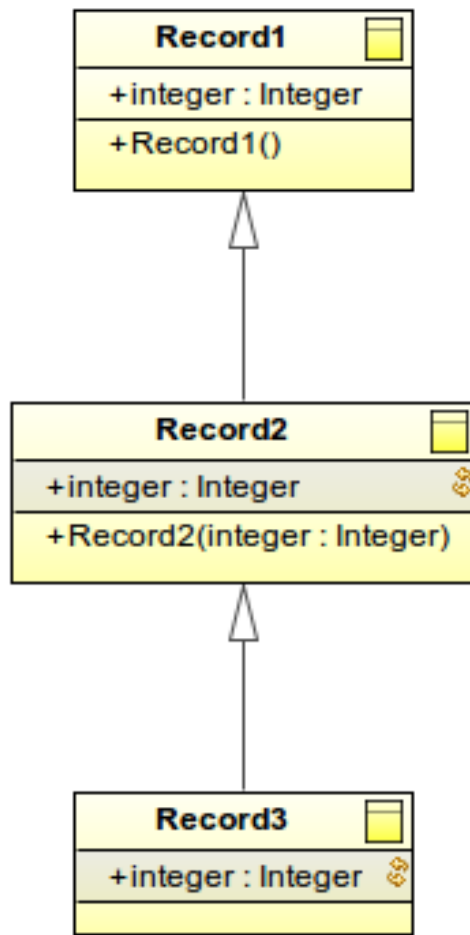


Figure 3.2 Record3 cannot be instantiated with the default constructor because Record2 declares a constructor so it is not possible to use its default constructor and it does not declare a non-parametric constructor

- Once you declare a constructor on your Record, neither the field constructor nor the default constructor is available.

Hence the constructors are available as follows:

- Once you declare at least one constructor, only the declared constructors is available.
- If you do not declare a constructor and a super Record contains a declared non-parametric constructor, the default non-parametric constructor is available
- When you do not declared a constructor and no default constructor is available, the field constructor is available.

3.5.5.2.2 Declaring Constructors

You can declare a constructor in method declaration using following the syntax:

```

public <YourRecordName> (<argument1_type> <argument_name>, <argument2> <argument_name>, ...) {
    <ConstructorBody>
}
  
```

The body of constructor is governed by the same rules as a method bodies.

Parametric constructor example:

```
Book {
    public Book(Integer isbn){
        super();
        //sets Field:
        this.isbn := isbn;
    }
}
```

To access fields of a Record, use the [access operator](#) `.`:

```
book.title
```

Note that the dot operator `.` fails with an exception if the `<EXPRESSION>` with the access operator is `null`. Use the [safe-dot operator](#) to prevent the exception.

When accessing a field or record that is of type Reference, the property is automatically dereferenced. Therefore the expression `(*ref).fieldName` and `ref.fieldName` are identical.

3.5.6 Property Path

Property Path holds a route from a Record to a Record Field, for example, `Employee.name`.

You can use Relationships to define a path to a Field of another Record, such as, `Employee.address.city` where `address` is the name of the Relationship to another Record, for example, `EmployeeAddress`.

To define the type:

```
PropertyPath
```

To create instances :

- to a Field via one or multiple relationships:

```
<Record>.<relationship>.<field>
```

```
MyRecord.myRecordsNavigableRelationship.fieldOfARelatedRecord
```

- to a Field on one Record:

```
<Record>.<field>
```

```
MyRecord.myRecordField
```

Note: To prevent the system from raising an exception when accessing a field that is `null`, you can use the `safe-dot ?.` operator. The operation then returns the value `null` if the reference expression is `null` (refer to [safe-dot operator](#)).

When accessing a Field or Record that is of type Reference, the property is automatically dereferenced. Hence the expression `(*ref).fieldName` and `ref.fieldName` are identical.

3.5.7 Property

The **Property** data type holds the type of a Field in a user-defined Record.

Note: Property is a special type of property path ([Property Path](#)): use PropertyPath type instead if possible.

Example

```
def Property authorNameField := Author.firstName;
```

3.5.8 Type

The **Type** data type holds data types, for example, a Type can hold the value String data type, a particular Map data type, a Record data type, etc. It is represented by *com.whitestein.lsp.lang.type.Type*.

The Type data type can be used to check if object are of a particular data type. The output can be then further used when [Casting](#) the object.

To define the type:

```
Type<T>
```

To create an instance:

```
//type is a keyword:
type(<TYPE>)
```

Example Usage

```
switch typeOf(person)
  case type(NaturalPerson) -> getFullName(person as NaturalPerson)
  case type(LegalPerson) -> getFullName(person as LegalPerson)
  case type(PersonGroup) -> getFullName(person as PersonGroup)
  default -> person.code + #" <Unknown type>"
end
```

3.5.9 Enumeration

An *Enumeration* is a data type that holds literals. Each literal is of the enumeration type. You cannot create an Enumeration type in the Expression Language directly: It is modeled as a *special Record type*.

To define the type:

```
<ENUMERATION>
```

To create an instance:

```
<enumeration_name>.<literal_name>
```

Example:

```
//Weekday is an enumeration with values MONDAY, TUESDAY, etc.
def Weekday wd := Weekday.WEDNESDAY;
```

You can compare enumeration literals with the comparison operators ==, !=, <, >, <=, >= for enumerations of the same enumeration type. Enumeration literals are arranged as a list of values; hence the comparison is based on comparing their indexes: the order depends on the order of the literal as modeled in the Enumeration.

In the LSPS Application, enumeration is represented by *com.whitestein.lsp.lang.exec.Enumeration* java class.

3.5.10 Null

The Null data type signalizes an unspecified value: its only value is `null`.

The data type is the subtype of every other data type, so that any object can take the `null` value.

To define the type:

```
Null
```

Chapter 4

Operators

Operators are symbols that cause a particular action, for example, comparison of values, summing up, assignment, etc.

4.1 Assignment Operator

To assign a value to use the assignment operator `:=`:

```
//declaring local variable x:  
def Integer x  
//assign value 1 to x:  
x := 1
```

4.2 Arithmetic Operators

Arithmetic operators are used on Integer and Decimal values similarly to their use in algebra.

Operator	Description	Example	Result	Note
+	addition	2 + 2	4	
-	subtraction	3 - 1	2	
++	increment by 1	intVar++ or ++intVar		postfix: returns value; then increments by one and assigns it to the referenced object (2++); prefix: increments value by 1 and reassigns it to the referenced object
--	decrements by 1	intVar-- or --intVar		postfix: returns value; then decrements by one and assigns it to the referenced object (2--); prefix: decrements value by 1 and reassigns it to the referenced object
*	multiplication	3 * 3	9	
/	division	9 / 3	3	
%	modulo	10 % 3	1	
**	exponentiation	3 ** 3	27	Since expressions are evaluated primarily in the left-to-right manner unless precedence or association rules take over, the expression 2**3**4 is evaluated as 2^3^4^4; the exponent follows the ** operator

4.3 Compound-Assignment Operators

You can use the following compound-assignment operators:

Operator	Description	Example	Equivalent
<code>+=</code>	assigns the result of the addition	<code>x += y</code>	<code>x := x + y</code>
<code>-=</code>	assigns the result of the subtraction	<code>x -= y</code>	<code>x := x - y</code>
<code>*=</code>	assigns the result of the multiplication	<code>x *= y</code>	<code>x := x * y</code>
<code>/=</code>	assigns the result of the division	<code>x /= y</code>	<code>x := x / y</code>
<code>%=</code>	assigns the remainder of the division	<code>x %= y</code>	<code>x := x % y</code>

4.4 Logical Operators

Logical operators are used to combine multiple expressions that each return a boolean value. Combination of expressions with logical operators return a single boolean value.

Operators and their return values

and, && (conjunction) true if both operands are true; otherwise false

or, || (inclusive disjunction) true when at least one of the operands is true; otherwise false

xor, exclusive disjunction true when the operands value is not identical, that is one operand is true and the other operand is false; otherwise false

not or ! (negation) true if the operand is false; false if the operand is true (unary operator)

Truth Table

Operand A	Operand B	A and B	A or B	A xor B	not A
true	true	true	true	false	false
true	false	false	true	true	false
false	true	false	true	true	true
false	false	false	false	false	true

Since logical expressions are evaluated from left to right, the short-circuit evaluation is applied on `and` and `or` expressions:

- for `and` expressions: `<false>` and `<not_evaluated>` evaluates to false
- for `or` expression `<true>` or `<not_evaluated>` evaluates to false

4.5 Comparison Operators

Comparison operators serve to compare two values. Comparing returns a boolean value.

To compare two values, you can use the following operators:

- `==` (equal) and `!=` (not equal, alternatively noted as `<>`) check if the values of any data type are equal.

```
def String varA:="value"
def String varB:="value"
varA==varB
//returns true
```

Note: When comparing records, it is the object identity that is compared, *not* the record value. Analogously, on shared records, the record IDs are compared. However, on non-shared records, you can define fields or relationships so that the values of these are used when comparing records (refer to the [Record Fields in the GO-BPMN Modeling Language Guide](#)).

- `instanceof` checks if the object is of a particular type.

```
def Integer int := 100;
int instanceof Decimal; //returns true

def Object obj := null;
obj instanceof String; // returns false
```

For a **null** argument, the operator returns `false`.

Note: The `core::isInstance()` returns `true` for **null** arguments.

- `<` (lesser), `>` (greater), `<=` (lesser or equal), and `>=` (greater or equal) check if values of *decimals*, *integers*, and *strings* are lesser, lesser or equal, greater, or greater or equal and returns `true` or `false`.

When comparing Strings, they are compared lexicographically as per Java lexicographic ordering of strings, for example, `"Čapek" > "Hemingway"` and `"Čapek" > "Asimov"` are true.

- `like` searches for an occurrence of a pattern in a string

The operator supports the wildcards `?` for one character and `*` for one or multiple characters

```
"matching exactly THIS word" like "* ?????? THIS *"
```

- `<=>` (the spaceship operator): checks if values are lesser, equal, or greater and returns `-1` if the left operand is lesser than the right operand, `0` if the left operand and right operand are equal, and `1` if the left operand is greater than the right operand.

Applicable to the String, Decimal, Integers, and Date types (Date is a complex data type defined in the Standard Library)

```
person1.name <=> person2.name
```

4.6 Concatenation

To concatenate two Strings, use the `+` concatenation operator.

```
varStringA+"String literal"
```

The String concatenation operator `+` can concatenate also a String and any subtype of the Object type: the Object type is converted to the appropriate String value and concatenated with the String; for example, a Date value is converted to a human readable date representation when concatenated with a string. Note that the String object must come first in such expressions.

String concatenation example If the first operands of the `+` operator is a string, the `+` operator is considered automatically the operand of concatenation and the non-string operand is automatically converted to a string.

```
"Timestamp: " + date("2014-07-31", yyyy-MM-dd)
//The date() function uses the DateTimeFormat implementation of the joda library ( http://www.joda.org/joda-api/)
//the function is evaluated as if wrapper in toString: toString(date("2014-07-31", yyyy-MM-dd))
```

Resulting string:

```
Timestamp: Thu Jul 31 00:00:00 CEST 2014
```

Note: If you want to keep a non-string operand before the string operand in concatenation, start your expression with the empty String literal "", for example, "" + author.surname + ", " + author.firstname.

4.7 Reference and Dereference Operators

The reference operator & returns [Reference](#) to a variable or a property path from a variable in the context of the variable. The dereference operator * takes a Reference value and returns the value currently stored in the referenced variable or property.

```
//instantiates record c1 with c1.name "Walter White":
def Partner c1 := new Partner(name -> "Walter White");
~
//reference variable ref with reference c1.name (current value "Walter White"):
def Reference<String> ref := &c1.name;
~
//variable x assigned dereferenced ref, that is "Walter White":
def String x := *ref;
~
//c1 assigned a new record instance with the name "Jesse Pinkman":
c1 := new Partner(name -> "Jesse Pinkman");
~
//new variable y assigned the dereferenced ref value (that resolves to "Jesse Pinkman"):
def String y := *ref;
~
//note that the value held by x remains "Walter White"
```

In the example the variable part is `c1` and the property part is `.name`. To acquire the value of a reference, use the dereference operator `*`.

References hold the referenced expression and the associated context. In our example the referenced expression is `c1.name`.

4.8 Inclusion Operator

The inclusion operator `in` checks if an element is in a set or list.

The check returns the String "1 is in mySet" if 1 is in mySet.

```
if 1 in mySet
  then "1 is in mySet"
end
```

4.9 Namespace Operator

The module namespace operator (: :) is used to refer to elements of other module namespaces.

Consider ModuleA with a variable var imported into ModuleB. You can access var from ModuleB as follows:

```
ModuleA::var
```

The mechanism of module import is described in the *GO-BPMN Modeling Language Specification*.

4.10 Selector Operator

To access items in collections or maps, use the selector operator [] to specify the element to be returned:

on sets

```
name_of_set[element_position]
```

on lists

```
name_of_list[element_position]
```

Note that the first element of a List is on position zero. For example, [10, 20, 30] [1] returns 20.

on maps

```
name_of_map[element_key]
```

4.11 Ternary Conditional

The ternary conditional operator ? : enables you to define a condition and two expressions. If the condition evaluates to `true` the first expression is returned. If the condition evaluates to `false`, the second expression is returned.

With the operator you can write the expression

```
if <CONDITION> then <EXPRESSION_1> else <EXPRESSION_2> end
```

as

```
<CONDITION> ? <TRUE_EXPRESSION> : <FALSE_EXPRESSION>
```

4.12 Null-Coalescing Operator

The null-coalescing operator `??` is a more effective version of `if <expression_1> != null then <expression_2>` with `<expression_1>` evaluated only once.

```
<expression_1> ?? <expression_2>;
//is equivalent to:
if <expression_1> != null then
    <expression_1>
else
    <expression_2>
end
```

Example:

```
def String title := getTitle() ?? "Default Title";
```

4.13 Access Operator

The dot operator `.` serves to [access fields of a Record](#), possibly via relationships, and a record's methods.

Example:

```
<EXPRESSION>.<FIELD>
```

For example:

```
book.title
```

Example:

```
<EXPRESSION>.<METHOD>
```

For example:

```
book.getAuthors()
```

Note that the dot operator `.` fails with an exception if the `<EXPRESSION>` with the access operator is `null`. When accessing fields, you can use the [safe-dot operator](#) to prevent the exception.

4.14 Safe-Dot Operator

To prevent the system from raising an exception when it attempts to access a record field of a record which is `null`, use the safe-dot operator `?..`

Similarly to the dot operator, the `?..` operator serves to access Record Fields, possibly of related Records. Unlike the dot operator, no exception is raised when the record is `null`. The expression simply returns `null`.

```
<expression>?..field
```

Example:

```
def Person person := null;
//returns null without an error:
person?.email;
//you can also chain the access requests:
person?.contact?.operator?.operatorCallCode
```

4.15 Cast Operator

To cast the value of an object to another type, use the `as` operator:

```
<objectName> as <newObjectType>
```

```
person as NaturalPerson
```

For further information, refer to [Casting](#).

4.16 Evaluation Order

The order of expression evaluation at runtime is generally from left to right, that is, first the left-hand operand is evaluated and only then the right-hand operand is evaluated.

The evaluation order can be influenced by the operator precedence. Mathematical operators, logical, and relational operators follow their natural operator precedence:

1. unary before multiplicative before additive and
2. negation before relational before equality before exclusive disjunction before conjunction before disjunction.

Parentheses override operator precedence and the expression in parenthesis is evaluated as a whole.

Operator precedence order

1. `::` (scope operator)
 2. `[]` (selector), `()` (function/closure call), `.` (dot operator), `?.` (safe dot operator)
 3. `+`, `-`, `&` (reference), `*` (dereference)
 4. `??` (ifnull)
 5. `*` (multiplication), `/` (division), `%` (modulo), `**` (exponentiation)
 6. `+` (addition), `-` (subtraction)
 7. `<`, `>`, `<=`, `>=`, `<=>`
 8. `instanceof`
 9. `==` (equal), `!=` (not equal), `<>`, `like`, `in` (inclusion)
 10. `cast`, `as`
 11. `not` and `!` (negation)
 12. `and` and `&&` (conjunction)
 13. `or` and `||`, `xor` (disjunctions)
 14. `?:` (ternary if)
 15. `:=` (assignment)
 16. `;` (chaining operator)
-

Chapter 5

Comments

Characters in code marked as comments are not interpreted on runtime and serve to provide information about the code.

To comment out a single line use the `//` characters: anything following the `//` characters until the end of the line is considered a comment.

```
uiCreateBook::createBook() //This is a comment.  
//The book is defined as a shared record.
```

To comment out multiple lines, mark the start of the comment with the `/*` symbols and finish it with `*/`

```
/* Multiline  
   code  
   comment */
```


Chapter 6

Local Variables

Local variables are created as part of an expression or an expression block and cannot be accessed from outside of it. However, from within an expression you can refer to any variable that exists in the scope of the expression and its parent scopes.

```
def String upperVar := "1";
begin
  def String lowerVar := "2";
  upperVar := "3";
end;
//this is not correct:
//lowerVar := "4";
```

To create a local variable use the `def` keyword in an expression. Note that `def` only declares the variable:

```
def <VARIABLE_TYPE><VARIABLE_NAME>
```

The variable value when declared is `null`, which is returned as its value.

To assign a value to a variable, use the assignment (`:=`) operator: such an expression *returns the right-hand-side value* of the assignment.

```
def String varString := "This is my variable value."
//returns "This is my variable value."
```

Note: In models, you can define also global and local variables. Global variables are accessible from the entire Model; local variables within the given resource, such as, a process or form. Mind you cannot create global or local variables in the Expression Language; these are created in dedicated `model resources`.

Chapter 7

Controlling Flow

7.1 Branching

Branching serves to accommodate different reactions depending on a particular condition.

You can perform branching using the appropriate `if` construct or a `switch`. Note that the constructs represent an [expression block](#).

7.1.1 if-then-end

The *if-then-end* returns the value returned by the `<expression>` if the `Boolean_expression` is true and `null` if the `Boolean_expression` is false.

```
if <boolean_expression> then
  <expression>
end
```

Example

```
//sendInfo is a Boolean variable.
if sendInfo then
  "Do send the newsletter."
end
//if sendInfo is false, the expression returns null (consider exception handling).
```

7.1.2 if-then-else-end

The *if-then-else-end* returns the value returned by `<expression_1>` if the `Boolean_expression` is true and value returned by `<expression_2>` if the `Boolean_expression` is false.

```
if <boolean_expression> then
  <expression_1>
else
  <expression_2>
end
```

Example:

```
//passedTest is a Boolean variable.
if passedTest then
  "Passed"
else
  "Failed"
end
```

7.1.3 if-then-elsif-end

- If the `boolean_expression_1` evaluates to false, `boolean_expression_2` is checked.
- If `boolean_expression_2` is true, `expression_2` is evaluated, and the evaluation leaves the if construct.
- If `boolean_expression_2` is false, the next *elsif* expression is checked, etc. If none of the *elsif* Boolean expression is true, `expression_N` is evaluated, and the evaluation leaves the if construct.

```
if <boolean_expression_1> then
  <expression_1>
elsif <boolean_expression_2> then
  <expression_2>
end
```

Example

```
//passedTest is a String variable.
if passedTest=="yes" then
  "Passed"
elsif passedTest=="no" then
  "Failed"
end
```

7.1.4 if-then-elsif-then-else-end

```
if <boolean_expression_1> then
  <expression_1>
elsif <boolean_expression_2> then
  <expression_2>
else
  <else_expression>
end
```

Example:

```
//passedTest is a String variable.
if passedTest=="yes" then
  "Passed"
elsif passedTest=="no" then
  "Failed"
else
  "Did not attend"
end
```

7.1.5 switch

The `switch` construct branches the execution flow based on condition value: it compares the argument expression against multiple possible values. If the value of the argument expression matches the value of the case, the expression defined for that case is executed and the `switch` returns the value of the expression. Unlike in Java, every case expression has an implicit `break`.

You can define a `default` expression that is executed if none of the cases matches is executed.

```
switch month
  case "January" -> 1
  case "February" -> 2
  default -> "Not January nor February"
end
```

7.2 Looping

Looping serves to repeat the same or similar action.

Note that the looping constructs represent an [expression block](#) block.

7.2.1 for

The *for* loop, server to loop through a block of expressions, until a condition becomes true.

```
for(init; condition; update) do
  expression
end
```

Example

```
def Integer i := 0;
for ( i; i < 10; i++) do
  debugLog({-> toString(i)}, 1000)
end
```

Important: Collecting results of *foreach*, *for*, and *while* in a collection so that you create a new collection on each iteration as below, is inefficient and can cause performance issues (consider that collections are immutable):

```
def Integer i := 0;
def Set<Integer> varSet := {};
for ( i; i < 10; i++) do
  //creates a new set with the i added and assigns it to varSet:
  varSet := add(varSet, i);
end
//varSet will contain { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }
```

Use *collect()*, *fold()*, *exists()*, *forall()*, etc. of the [Standard Library](#) instead. For example:

```
collect(1..10, { x:Integer -> new Option(label -> "Option " + x, value -> "Value " + x) })
//instead of:
//def List<Option> options := [];
//foreach Integer x in 1..10 do
//  options := add(options, new Option(label -> "Option " + x, value -> "Value " + x) )
//end;
//options;
```

7.2.2 foreach

To iterate through items in a collection, use *foreach*:

```
foreach <type> <iterator_name> in <collection> do
  <expression>
end
```

Example:

```
def Set<Person> persons := { ... };
~
foreach Person person in persons do
  sendEmail("Important Notification", "", {}, {person.email}, {}, {}, "UTF-8");
end
```

7.2.3 while

To loop code while an expression is true, use the `while` construct:

```
while <boolean_expression> do
  <expression>
end
```

7.2.4 break

In *while*, *for*, and *foreach* loops, you can use the *break* keyword to finish the looping immediately and continue with the next expression.

```
def Integer i := 0;
for ( i; i < 10; i++) do
  if i == 3 then
    break;
  end
end
```

7.2.5 continue

In *while*, *for*, and *foreach* loops, you can skip the current loop with the *continue* keyword.

```
def Set<Person> persons := {};
~
foreach Person person in persons do
  if isEmpty(person.email) then
    continue;
  end;
  sendEmail("Important Notification", "", {} , {person.email}, {}, {}, "UTF-8");
end
```

Chapter 8

Exception Handling

On runtime, code can cause an error that halts the execution and potentially terminate the execution unexpectedly. Typically, this can occur on user input, when the user input is unexpected (for example, while the code expects a Decimal value and the user enters a value with letters). Exception handling enables you to deal with such situations and handle the thrown exception gracefully.

To handle an exception, use the `try-catch` construct on the code.

Also, you can decide that a particular expression should produce an exception. To throw an exception, use the `error()` function from the Standard Library.

8.1 Throwing Exceptions

The `error()` function throws an error with an error code. The error code is a String parameter of the construct.

```
error(<errorcode>)
```

Example:

```
error("InvalidISBNFormat")
```

If an error exception is not caught and handled, the execution terminates. The error can be caught and handled by a try-catch block.

Important: Error throwing functions are part of the Standard Library. Refer to the Standard Library Reference for further information.

8.2 Catching Exceptions

To catch and handle error exceptions produced by `throw` or by null function parameters which must not be null without interrupting the execution, use the try-catch statement block on the code which might cause an exception:

```
try <expression>
  catch <error_code>, ... -> <handle_expression_1>;
  handle_expression_2;
  ...
end
```

If the `catch` statement takes **null**, any error is caught.

Example:

```
try getCode()
  catch "Invalid ISBN format", "Invalid ISSN format" -> "Code value is not valid."
end
```

Example: Catching any error code

```
try getCode()
  catch null -> "Code value is not valid."
end
```

Note that the block returns an object, which you can cast as appropriate:

```
try val.toDecimal()
catch null -> "not decimal" as String
```

8.3 Built-in Errors

The *Expression Language* makes use of the errors with the following error codes:

Error	Description (occurrence circumstances)
AmbiguousNameError	The provided name cannot be resolved to a unique entity.
ArithmeticError	An operand in an arithmetic operation cannot be processed.
BinaryDataError	Binary data cannot be retrieved (for example, from database).
DoesNotExistError	The entity does not exist.
FormatError	The format of an argument is incorrect (for example, on casting of a String).
IncompatibleTypeError	The type of processed value is incorrect (typically on casting or assigning).
IncorrectPathname	The string with the path is invalid.
MergeEvaluationError	Evaluation context created by a View Model cannot be merged.
ModelInstantiationError	Instantiation of a model failed.
ModelInterpretationError	Model cannot be interpreted.
NoExternalRecordProvider	The resource with the requested external record is not available.
NoSuchPropertyError	The record property does not exist.
NullParameterError	A mandatory parameter has the <code>null</code> value.
OutOfBoundsError	The collection element does not exist.
OptimisticLockConflictException	Saving changes to record instance failed since the underlying versioned record instance was changed.
ReadOnlyAccessError	The system attempted to write to a read-only object.
RecordNotFound	The record instance does not exist.
ReferenceNotFoundError	Reference is failed (Reference does not exist).

Chapter 9

Functions

A function is a named [closure](#): it has input parameters which are passed to the body of the function. The body comprises one or multiple expressions and return a value, which is the return value of the function.

Functions are declared and defined in a function definition file.

Note: PDS comes with two function editors, the text function editor and the visual function editor, which serve to create two different types of function definition files: When using the visual function editor, the output is an XML file with the declarations and definitions. The text function editor allows you to declare and define your functions as text. This section deals exclusively with the syntax of functions when they are defined as text. For more information on how the functions are declared and defined in PDS, refer to [Functions in PDS guide](#).

To define a function use the following syntax:

```
/** <description> */
<annotations>
<visibility> <returnType> <name> (<parameter1_type> <parameter1_name>, <parameter2_type> <parameter2_name>, ... )
    <implementation>
}
```

Methods define their **visibility**:

- *private*: accessible only from within the Module
- *public*: accessible from the Module and from any importing Modules

Function parameters can be flagged as:

- mandatory using the * symbol (unflagged parameters are optional)
- variadic using the ... after the parameter type, for example, `String... name`

To define the default value of a parameter, use the `<PARAM_TYPE> <PARAM_NAME> := <DEFAULT_PARAMETER_VALUE>`

9.1 Functions Implemented by a Java Method

When implementing a function as a Java method, use the **native** keyword followed by the java method path that implements the function instead of the {<implementation>}, for example, `native com.whitestein.lsp.example.myfunction;`

Note: The native implementation then receives the current context as its parameter.

Individual functions are declared and defined with the following syntax:

```
/** <description> */
<annotations>
<visibility> <returnType> <name> (<parameter_type> <parameter_name>)
native <implementing_method>
```

9.2 Function Annotations

A function declaration can have the following annotations:

- **@SideEffect**: on validation, the info notification about that the called function has a side effect is suppressed
A function is considered to have side effects if one of the following is true:
 - The function modifies a variable outside of the function scope.
 - The function creates a shared record.
 - The function modifies a record field.
 - The function calls a function that causes a side effect.
- **@Deprecated**: on validation, a notification about that the called function is deprecated is displayed
- **@Disabled**: function is disabled (no calls can be performed)
- **@Status**: `modeling status` of the function as `@Status (<STATUS_NAME>)`
- **@Meta**: metadata of the function defined as `@Meta(key1 -> value1, key2 -> value2, ...)`
- **@ExtensionMethod**: `extension methods` of existing Records; the first parameter is the Record you are extending

9.3 Calling Functions

Function calls are calls to function definitions which are special kinds of closures defined in a function definition: Function definitions are model elements which cannot be created directly in the Expression Language; however, you can call functions and use their return value in your expressions.

A function call follows the syntax

```
<FUNCTION_NAME> (<COMMA_SEPARATED_ARGUMENTS>)
```

or alternatively

```
<FUNCTION_NAME> (<PARAMETER_NAME_1> -> <ARGUMENT_1>, <PARAMETER_NAME_2> -> <ARGUMENT_2>)
```

The explicit definition of parameter names is useful if a parameter is not required and has a default value.

Example function call:

```
getModel("Delivery", 1.4)
//alternatively:
getModel(name -> "Delivery", version -> "1.4")
```

If a function uses **type parameters**, their types are inferred. However, you can define the types explicitly if required:

```
<FUNCTION> | <COMMA_SEPARATED_TYPES_PARAMETER_TYPES> | (<ARGUMENTS>)
```

The list of types in `<COMMA_SEPARATED_TYPES>` is used in the same order as the type parameters are defined. Note that you need to define the types for all type parameters.

A function call is resolved into the function based on the call arguments: overloading is supported.

Call to a Standard Library function with the types of type parameters:

```
//collect has the E and T type parameters:
//E will be handled as Employee and T as Decimal:
sum(collect|Employee, Decimal|(e, {e -> e.salary}))
```


Chapter 10

Model Elements

Expressions can access named elements defined in the model, which cannot be defined directly in the Expression Language.

Such model elements include the following:

- **Modules** represent a structuring unit similar to a package and contain all the resources with model elements. A module can use resources of another module only if it imports the module—similarly to packages in Java. The module importing mechanism is described in the [GO-BPMN Modeling Language guide](#). If you want to reference an element from another module, the name of the entity must be preceded by the module name and the :: operator.

```
<MODULE_NAME> :: <ELEMENT_NAME>
```

- **Queries and form** are declared and defined in their definition files and are called just like functions:

```
<QUERY_OR_FORM_NAME> (<PARAMETER_1>, <PARAMETER_2>)
```

Note that parameters can be themselves expressions that return the data type defined as the parameter type. If a parameter is of the String type, the parameter can be an expression that results in a String object.

```
getName (getProcessName () + "#" + getId ())
```

Note: Other elements are implemented as functions and are called in the same way. This includes queries and forms.

- **Variables from parent namespaces** are defined in a variable definition file (global module variables), or directly on the elements representing the parent namespace, for example, a sub-process, a form, etc. Variables from parent namespaces are referred to by their name with no special notation.

- **Records and their fields** are defined in a data type definition file.

Records are referred to by their name with no special notation. To access record fields, use the dot operator: `\<RECORD_NAME\>.\<RECORD_FIELD\>`

Note that Record properties, fields on related Records, are accessed in the same way.

- **Constants** are named values of the enumeration data type, or maps of these data types. After their value has been initialized, it remains unchanged during the rest of the runtime. Initialization expressions of constants can use other constants.

Constants cannot be defined in the Expression Language directly. To call a constant in an expression, use its name, for example, `"This is the current date format: " + DATE_FORMAT.`

Chapter 11

Reserved Words

The following words are reserved words intended for future use:

- `final`
- `repeat`
- `until`
- `return`

