

Fuego Business Language (FBL) 5 Programming Guide

Fuego, Inc.

Fuego Business Language (FBL) 5 Programming Guide

by Fuego, Inc.

Published January, 2005 - Version 5.5. Revision 10 - June, 2006.

Copyright © 2001-2006 Fuego, Inc.

Fuego Business Language (FBL) 5 Programming Guide

Copyright 2001-2006 Fuego, Inc. All rights reserved.

This documentation is subject to change without notice. This documentation and the software described in this document contains proprietary trade secrets and confidential information of Fuego, Inc. and is also protected by U.S. and other copyright laws and applicable international treaties. Use of this documentation and the software is subject to the license agreement between you and Fuego, Inc. If no such license agreement exists, you may not use this documentation and software in any manner whatsoever. Unauthorized use of the documentation or software, or any portion of it, will result in civil liability and/or criminal penalties. U.S. Patent Pending.

Fuego, Fuego 4, Component Manager, Process Designer, Work Portal, Orchestration Engine, Execution Console, Process Analyzer, Organization Administrator are trademarks or registered trademarks of Fuego, Inc.

FuegoBPM 5, FuegoBPM Studio, FuegoBPM Designer, FuegoBPM Enterprise Administration Center, FuegoBPM Work Portal, FuegoBPM Portal Console, FuegoBPM Archive Viewer, FuegoBPM Logviewer, FuegoBPM Express Server, FuegoBPM Enterprise Server, FuegoBPM Application Server Edition, FuegoBPM Web Console, FuegoBPM Process Analyzer, FuegoBPM Data Store, FuegoBPM Dashboard, FuegoBPM BAM, FuegoBPM Portlets, FuegoBPM Suite, FuegoBPM Deployer, FuegoBPM Failover, FuegoBPM VCS, FuegoBPM Ant Tasks, FuegoBPM FDI, FuegoBPM Help Viewer, FuegoBPM Server are trademarks or registered trademarks of Fuego, Inc.

InstallAnywhere is a registered trademark of Zero G Software, Inc. Solaris and Java are trademarks of Sun Microsystems, Inc. Windows is a registered trademark of Microsoft Corporation.

All other trademarks, trade names, and service marks are owned by their respective companies.

Table of Contents

1. FuegoBPM Basics	8
Business Services Orchestration	8
What's FuegoBPM	11
2. Fuego Methods Introduction	14
Kinds of Methods	14
Comments	15
Programming Styles	16
3. Variables	20
Names, declaration and Scope	20
Initialization	23
4. Components	25
Creating objects	25
Current and Default instance	27
Duplicating objects	28
Calling members	29
Cleaning Up Objects	31
5. Expressions	32
6. Operators	35
Arithmetic operators	35
Relational operators	36
Logic operators	36
Other operators	38
7. Numbers	40
Real vs Decimal numbers	44
Decimal Arithmetic	45
Number Functions	50
8. Strings	67
String Functions	69
String Attributes	82
9. Times and Intervals	83
Time Attributes	87
Interval Attributes	100
Time Functions	108
Interval Functions	133

10. Arrays	142
Indexed Arrays	143
Associative Arrays	146
Manipulating arrays	148
Array Functions	151
Array Attributes	157
Array Procedures	158
Mapping members	160
11. Enumerations	161
12. Statements	163
Interactive Statements	163
Input Examples	175
Long running statements	181
Control Statements	184
Compound Statements	185
Conditional Statement	187
Multi Path Conditional Statement	190
Bounded Loop	192
Unbounded Loop	194
Exit statement	195
Labeled Statements	197
Throw Statement	198
Assignment	199
Logging Statement	201
Type conversions	202
Transformations	204
13. Exceptions	205
ClassNotFoundException	206
NumberFormatException	207
NegativeArraySizeException	207
ArithmeticException	207
CloneNotSupportedException	207
NullPointerException	208
ArrayStoreException	208
IllegalThreadStateException	208
StringIndexOutOfBoundsException	208
Exception	208
ArrayIndexOutOfBoundsException	208

IllegalAccessException	209
UnsupportedOperationException	209
InterruptedException	209
InstantiationException	209
IllegalArgumentException	209
IndexOutOfBoundsException	210
RuntimeException	210
SecurityException	211
IllegalMonitorStateException	211
ClassCastException	211
IllegalStateException	211
14. Embedded SQL	212
INSERT statements	212
DELETE statements	213
UPDATE statements	214
SELECT statements	215
Referencing columns	217
Aggregate functions	219
Positioned updates and deletes	221
SQL operators	222
SQL keywords	225
Stored procedures	226
15. Regular Expressions	227
Functions	227
Simple Matching	229
Modifiers	229
Metacharacters and Characters Sets	230
Matching repetitions	232
Anchors	235
Alternations	238
Grouping	239
Extraction	240
Backreferencing	246
Search and Replace	247
16. Code Conventions	249
General Naming Conventions	250
Specific Naming Conventions	253
Statements	257

Layout And Comments	261
17. Refactories	268

Chapter 1. FuegoBPM Basics

Business Services Orchestration

The FuegoBPM (TM) Suite embraces and extends the concept of Business Process Management (BPM) through its vision of Business Services Orchestration (BSO.)

BPM is a discipline that includes many different types of tools and methodologies. A simple process modeling tool, such as Visio, can be considered a BPM utility. Business Intelligence tools can be considered BPM utilities. True, in today's market more people are starting to see BPM as a new category of software that **automates business processes**. The problem is: what do we really understand by automating business processes?

- For the creators of BPEL, it is the organization in time of web services invocation
- For EAI fans, it is a state server that coordinates messages on a proprietary bus
- For some ERP vendors, it is the business logic embedded in an ERP system
- For traditional workflow vendors, it is the organization of the collaboration between people

FuegoBPM can be used to fit in any of the above visions, but they fall short of what FuegoBPM was meant to do.

For FuegoBPM, automating business processes consists of **managing the behavior of people, systems and organizations to orchestrate a repeatable business service**.

Therefore,

- FuegoBPM sees organizing the invocation of web services as managing the behavior of systems, and not all systems: only those exposed as web services.
- FuegoBPM sees a state server to coordinate messages as managing the behavior of systems, and not all systems: only those that have adapters into a proprietary messaging bus.
- FuegoBPM sees the business logic embedded in an ERP system as a service that manages the behavior of organizations limited by the rules in the ERP system. This service can be reused in the context of a cross application enterprise process.
- FuegoBPM sees the organization of the collaboration between people as managing the behavior of people.

Fuego's vision of BPM includes all the above visions in one single holistic vision: Business Services Orchestration. FuegoBPM sees anything a person, system or organization does within an enterprise as a **Business Service**. FuegoBPM provides all the necessary tools to **Orchestrate** composite business services using existing ones, manages and measures the service levels of those composite business services and continuously improves them.

This is what we call *Full Lifecycle Management of Orchestrated Business Services*.

To be able to do this, FuegoBPM provides the full set of tools that enables companies to:

1. Model Processes.
2. Transform Process Models into executable designs.
3. Simulate the execution of designs to study the feasibility of a service level.

4. Harmonize and catalog business services from existing systems to be able to use them regardless of what technology is used to expose them.
5. Catalog the different services from people that can be rendered by the organization and their availability in time.
6. Expose composite services that orchestrate services from systems people and organizations to be reutilized.
7. Monitor the orchestration in production according to the parameters set forth in the simulation.
8. Measure the performance of the process from a historical perspective.
9. Use statistical data to refine future simulations.

FuegoBPM can be used to manage the full spectrum of business processes, from the mostly automated (like BPEL) to the more collaborative processes like those that involve specialized workers and creative activities.

When designing with FuegoBPM, it is critical to understand that the Server was conceived to manage **behavior** rather than just to pass data. When working with a business service, the invocation of the service provokes behavior, when presenting a user with a work portal, the Work Portal suggests the adequate behavior to the user. Obviously, the user is free to do as he or she wishes, but it is very convenient not to need to remember the adequate behavior in each intervention in each process in which a user is involved. And, whatever gets done in effect by people, systems and organizations is logged into a process log that allows the tracking, tracing and measuring of performance.

Without any doubt, Business Services Orchestration is the most complete way to automate the management of a business process designed, for example, as a result of a six sigma exercise, ISO

compliance exercise or BPR exercise. Why?

Because the FuegoBPM Enterprise Server will elicit behavior that otherwise would have implied months of training and convincing, and eons of application integration.

Moreover, Business Services Orchestration is the easiest way to build composite apps that integrate existing ones and expose them as web apps or web services.

To provide the ideal Orchestration platform FuegoBPM has centralized all the design and development tools in a single environment: FuegoBPM Studio. As well the design can be previously defined in the FuegoBPM Designer and the development can be completed using the FuegoBPM Studio.

The orchestrations created in Studio run on an orchestration server that comes in two categories: Express and Enterprise.

The Express category of servers is designed for quick deployment of departmental and small business orchestrations that will require no administration or for proof of concept projects in their pre-rollout stage.

The Enterprise category of servers is designed for full featured Enterprise security, scalability and failover capabilities as well as to run inner-departmental and inter-enterprise processes.

What's FuegoBPM

FuegoBPM is a full-life cycle development and runtime environment for managing business processes from a Business Services Orchestration (BSO) perspective. This means that FuegoBPM focuses on managing the behavior of people, systems and organizations (through a process metaphor) to fulfill a measurable and repeatable business service that may span departments, divisions and company boundaries.

The full-life cycle development environment is FuegoBPM Studio.

Studio provides all the necessary functionality for a BSO approach towards BPM.

The full-life cycle runtime environment is provided through two runtime server editions:

- FuegoBPM Express - an entry level server that requires zero administration, fit for self-contained business services or for proof-of-concept projects.
- FuegoBPM Enterprise - the full fledged enterprise edition to run processes that span departments, divisions and enterprises with all the scalability, security and flexibility features you would expect from an enterprise grade product.

FuegoBPM caters to the needs of our customers in terms of TCO (Total Cost of Ownership) and ROI (Return on Investment). This is why we can really improve the way businesses run. FuegoBPM helps businesses increase operational efficiencies, reduce costs and increase profitability with an agile BPMS that can adapt to any budget and manpower. FuegoBPM allows companies to take control and tangibly optimize enterprise assets—applications, people and core business functions – and how they work together. With FuegoBPM, companies can quickly fill the gap between business strategy and execution in order to gain immediate payback.

FuegoBPM provides a BMPS software that makes the critical enterprise assets work the way you do and change as you change. By orchestrating applications, people and partners into executable, end-to-end processes that can be exposed as new composite business services, FuegoBPM fills the gap between business strategy and business execution.

FuegoBPM shields the process logic from the differences that arise from location (timezone, holidays, vacations, language), from IT infrastructure (MS, Unix, Legacy), from IT strategy (J2EE, .NET, Websphere, CORBA) and from the applications that contain reusable

services (SAP, Peoplesoft, I2, Siebel, legacy, etc.). Therefore, allowing non-specialized business analysts to model, design and change processes with no need to be domain experts.

FuegoBPM reduces complexity, enhances productivity and makes any company as competitive as its creativity allows (not limiting process automation to that which their enterprise software vendors provide.)

Chapter 2. Fuego Methods Introduction

Fuego Business language (FBL)

FuegoBPM Studio provides a high-level language used to define business rules and logic for activity types and certain transitions within a process. This language has been designed to easily integrate systems and to clearly express business logic. It also has some advanced features, such as:

- Programming Styles
- An advanced editor that supports syntax coloring, code completion, templates, just-in-time error checking, and so on.
- Seamless integration with different technologies including COM, CORBA, EJB, Java, SQL, XML, Web Services, and so on.
- A large library of built-in components.
- Built-in support of regular expressions.
- Others.

Kinds of Methods

In FuegoBPM Studio, there are two kinds of methods:

- Business Process Method
- Business Object Method

Both kinds are very similar in the features they support, but they differ in:

- Their visibility.
- The set of available predefined variables.
- Their runtime environment.

Business Process Methods can only be accessed from the process to which they belong. They are usually the implementation of an activity. They have several process-related, predefined variables and they always execute inside a process controlled transaction.

Business Object Methods can run on the server side or the client side. They can be defined as functions and inherit behavior from a superclass of the object that contains them. Typically, they are visible from the entire project.

Comments

Comments are one of the most important things to remember when writing a method. Including comments makes it easier for you and other developers to read a method. Comments can be single line or multi-line.

Single line comments are denoted with a double forward slash (//).

```
// This is a single line comment
```

Multi-line comments are enclosed between a forward slash and an asterisk (/*) and an asterisk and a forward slash (*/).

```
/* This is a multi-line comment. It can span multiple lines  
   of code and can be as long as you want it to be.  
   You do not have to worry about line breaks, although  
   you may add them if you want to. */
```

Note



VB style uses an apostrophe (') for single line comments.

Using Comments

Always remember that when you write comments, you should explain the *why* and not the *how*. The *how* can be read from the code itself. For further information on suggested practices, please see: Code Conventions

Programming Styles

FuegoBPM Studio supports different programming styles to reduce the time needed to learn how to program business process methods.

Each style mimics a well-known programming language as precisely as possible and adds the features that are required to write your business rules effectively.

In FuegoBPM Studio, three programming styles are available:

- Fuego: The traditional Fuego Business Language (FBL) syntax with its English-like construction for self-documenting code.
- Java: The popular run-anywhere programming platform.
- VB: Emulating Microsoft's Visual Basic .Net syntax.

Fuego Style

This is the traditional style (and the recommended one) for Business Process Methods.

The following example shows some of the characteristics of the FBL programming style:


```
firstName as String
lastName as String
selectedButton as String
// Ask the person's name
input "First Name:" : firstName,
      "Last Name:" : lastName
      using title = "Enter Your Name",
          buttons = ["Done", "Cancel"]
      returning selectedButton = selection

// Check the button pressed
if selectedButton = "Done" then
    display "Hello " + firstName + "!"
else
    display "I'll just call you John"
end
```

Java Style

Emulates Java syntax and adds several features to match the expressiveness of the Fuego style, such as:

- Output arguments.
- Input and display statements.
- Variable auto-initialization.
- Transformations.

```
String firstName;
String lastName;
String selectedButton;
// Ask the person's name
input("First Name:" firstName,
      "Last Name:" lastName,
      title : "Enter Your Name",
      buttons : { "Done", "Cancel" },
      out selection : selectedButton);
```

```
// Check the button pressed
if (selectedButton == "Done") {
    display("Hello " + firstName + "!");
}
else {
    display("I'll just call you John");
}
```

VB Style


This style emulates Microsoft Visual Basic .NET syntax. Note that unlike Visual Basic. Net, the VB style is case sensitive and it also has several features that Visual Basic .Net does not, such as:

- Input and display statements.
- Variable auto-initialization.
- Transformations.

```
Dim firstName As String
Dim lastName As String
Dim selectedButton As String
' Ask the person's name
Input "First Name:" : firstName,
      "Last Name:" : lastName,
      title := "Enter Your Name",
      buttons := { "Done", "Cancel" },
      Out selection := selectedButton

' Check the button pressed
If selectedButton = "Done" Then
    Display "Hello " & firstName & "!"
Else
    Display "I'll just call you John"
End If
```

Note

 Trademarked names appear throughout this documentation. Instead of list names and entities that own the trademarks or insert a trademark symbol with each mention of the trademarked name, Fuego states that it is using the names only for editorial and description purposes and to the benefit of the trademark owner with no intention of infringing upon the trademark.

Chapter 3. Variables

Variables

Variables are locations in memory (and sometimes in a database) in which any value can be stored. Each variable has a name, description, type, and value. Most variables only store data of one particular type. For example, if you define a variable to store a string, it would not normally be used to calculate integers.

Because data types are commonly fixed, the compiler will check to ensure that you are using correct variable types in your Method. If a Method statement is meant to process integers, the compiler will detect when you inadvertently try to use it to process another type of data, such as a string.

For information on how to use the different types of variables from the FuegoBPM Studio, please refer to the following topics:

- Using Variables
 - Instance Variables
 - External Variables
 - Argument Variables
 - Predefined Variables
 - Local Variables

Names, declaration and Scope

Valid variable names must start with an alphabetic character or an underscore character followed by zero or more alphanumeric

characters. For example, the following are all valid variable names:

```
participantName = "John Doe"  
iso9001 = "..."  
_ugly_variable_name_ = 1
```

On the other hand, the following are all invalid (in Fuego style):

```
//Variables cannot start with numbers  
4ever = true  
//display is a reserved word  
display = true
```

There are certain names that cannot be used as variable names. These are called reserved words. Reserved words are dependant on the current language skin. To use a reserved word as a variable name, you must escape it with the @ character, for example:

```
//This is legal, because the reserved word has been escaped  
@display = true  
//References to it must also include the @ sign  
display @display
```

Note that the @ sign is not part of the variable name. It is just a way to tell the compiler that you are not referring to the reserved word but that you want to refer (in this case) to the variable.

Variable Declaration

The variable declaration syntax is dependant on the skin you are using, The following examples declare two variables:

- **name** of type **String**
- **temp** of type **Any**

In Fuego style:

```
name as String
temp as Any
```

In Java style:

```
String name;
Any temp;
```

In VB style:

```
Dim name As String
Dim temp As Any
```

Each of the above declarations has a 'local' scope.

Scope

The scope of an entity is the part of a project that can gain access to such entity by name (here, an entity can be a method, a variable or a component).

Here are some examples:

1. Catalogued components can be accessed from anywhere in the

project.

2. A process' instance variable can be accessed from any part of the process it belongs to, except global activities.
3. A method that implements a Global Activity can be used from any part of a process (it is a static method).
4. A local variable can only be accessed from the method that declares it and all its inner blocks.

Entities with narrower scopes shadow the ones declared in an outer scope. For example: argument variables shadow instance variables.

Certain scopes such as the one defined for instance variables or the one for argument variables have special qualifiers that let you disambiguate the entity to which you are referring:

```
// Suppose you have an instance variable named 'name'
// and you declare:
name as String
// To assign to your local variable the value of
// the instance one, you must do:
name = this.name
// or you may also have an argument named 'name':
name = arg.name
```

This may come in handy because sometimes scopes overlap (for example, local variables and argument variables).

Initialization

Default Values

In FuegoBPM Studio, all variables are automatically initialized when first used if there is a suitable default value for the variable's type.

The following table summarizes the default values that are used for each type:

<i>Type</i>	<i>Default Value</i>
Numeric Types	0
String	""
Any	null
Time	'now'
Interval	'0s'

Initialization

Variables can be initialized in the declaration:

```
name as String = "Hello"
```

or after the declaration by using an Assignment Statement:

```
name as String  
name = "Hello"
```

Chapter 4. Components

Using Components

FuegoBPM Studio makes extensive use of components. It includes a large library of built-in components for common tasks. You can write your own components inside FuegoBPM Studio (Fuego Objects) and you can include different technologies as components in the component catalog.

Components define a type, which can be used to declare variables. All components can be used to declare a local variable or an argument variable, but not all components can be used as instance variables.

This happens because instance variables are usually persisted (a process' instance variable) or transferred (a Presentable Fuego Object instance variable). And, for the persistence or instance variables transference to work, the content of such variables must support serialization. Certain components do not support serialization.

A component can be identified by the casing. Its name always starts with an uppercase letter. For further details, refer to the General Naming Conventions topic.

For further information on components usage in FuegoBPM Studio, please refer to the following topics:

- [Implementing Business Objects using Fuego](#)
- [Introducing Business Objects into Fuego](#)

Creating objects

As mentioned in Variables, all variables have a type.

Variables of primitive types (such as `String`, `Int`, `Real`, and so on) always have a default value. On the other hand, variables that have a non-primitive type have special rules of initialization (as seen in [Variable Initialization](#)).

In this section, we will discuss how to explicitly initialize non-primitive variables.

Constructors

For initialization, we can group components in two categories:

- Instantiable components
- Components that must be obtained from another component

The difference between the two categories is that the former has a special method called the **constructor**, which is used to create new instances of the component. The latter does not.

Constructors are methods that are named after the component, which may or may not have arguments. If the constructor of a component does not have arguments, it is called the **default constructor**.

The syntax to initialize a variable is the following:

```
variable = [Module.] ComponentName ( [ [ {argument name} :]  
                                     {value} [,...] ] )
```

Note that the names and types of arguments depend on the component and on the constructor you are calling.

Consider the following example:

```
configFile as TextFile
configFile = TextFile(name : "/home/config.props")
for each line in configFile.lines do
    //Process the lines...
end
```

In the example above, on the first line, a local variable named *configFile* of type *TextFile* is declared, and on the second line it is initialized using "TextFile's" constructor, passing the file's name as an argument.

Current and Default instance

FuegoBPM Studio has the concept of a default instance, which is an instance associated to a component.

Only components that have default constructors have default instances. And such instances are accessible within the method scope. That is, a default instance that has been created while running a specific method only exists through that method's execution.

For example:

```
show Menu
    using entries = ["Apple", "Oranges",
                    "Chocolate"],
        title = "What you like best?"
```

In the example above, the reference to the component *Menu* is using its default instance, that is, an instance is automatically created the first time that a reference to Menu appears.

Current Instance

Typically, when you want to refer to your current instance, you use

the keyword **this** (or **Me** in VB style):

```
update this using date = 'now'
```

Suppose that the above-mentioned code belongs to a component named *MyComponent*. You could write the code above as:

```
update MyComponent using date = 'now'
```

What has happened is that when you refer to the default instance of a component and that component is the same as **this**, the current instance is used as the default instance.

Duplicating objects

Sometimes you want to create an exact copy of an object.

Note that assigning a component to a variable **does not** create a copy of it. In order to do so, you can use the **clone** function:

```
//Create an instance for each participant in the role
for each person in activity.role.participants do
  copy = clone(this)
  copy.participant.next = person
end
```

Clone Behavior

The clone function behavior depends on the object implementation. It executes following the next steps:

- If the object you are trying to clone has a method named clone

and implements the interface Cloneable, that method is used to obtain a copy.

- If the object implements the Serializable interface, it attempts to serialize it and deserialize it to obtain a copy of the object.
- Otherwise, it attempts to dynamically create a copy of it.

Calling members

FuegoBPM Studio supports three syntaxes for calling methods. The other styles have their own that are inherited from the language they are trying to mimic. The three syntaxes are the following:

- Verbose
- Object-method
- Functional

Verbose

The Verbose calling syntax is as follows:

```
method Component
[ using
    argName={value} [, ...]
]
[returning
    variable = outputArgument [, ...]
]
```

The following example uses the verbose syntax to display a Menu:

```
choice as String
show Menu
    using entries = ["Apple", "Oranges",
                    "Chocolate"],
        title = "What you like best?"
    returning choice = selection
```

Object-method

This syntax is the classic object-oriented syntax, which can be summarized as follows:

```
Object.method([[out] argName:
               {variable or expression}[,...]])
```

The same Menu code example using the object-method syntax is:

```
choice as String
Menu.show(
    entries: ["Apple", "Oranges",
             "Chocolate"],
    title: "What you like best?",
    out selection: choice)
```

Functional

This syntax is intended to be used on methods that act as functions. The following summarizes this syntax:

```
function(Object, [[out] argName:
                  {variable or expression}[,...]])
```

The same Menu code example using the object-method syntax is:

```
choice as String
show(Menu,
    entries: ["Apple", "Oranges",
              "Chocolate"],
    title: "What you like best?",
    out selection: choice)
```

Cleaning Up Objects

FuegoBPM Studio automatically releases the memory used by components when they are no longer used. It is not necessary to 'release' or 'clean' the components used in a method. However, there are certain components that require some kind of cleaning before ending the execution. They must be cleaned by using an 'exit' block to ensure that they are always cleaned up. For example:

```
do
    // use components here
on exit
    // clean used components here
end
```

Note that the code enclosed in the **on exit** (Java style: **finally**, VB style: **Finally**) part of the block is executed even if an exception occurs. Next, after the execution, the original exception is thrown, unless it is masked by an exception thrown in the **on exit** block.

Chapter 5. Expressions

Expressions

Expressions are operations in algebraic format that yield a value when evaluated. Expressions have no side effects, which means that if the variables in an expression do not change, the result of the evaluation remains the same.

An expression consists of operators and operands. Operators are special symbols commonly used in expressions. Operands can be variables or function invocations. However, since the expression cannot have side effects, the function invocation cannot have side effects either.

Expressions must operate on homogenous variable types. Type checking is performed at compile time to guarantee that no runtime errors occur due to an invalid mix of types.

Examples of expressions are as follows:

```
// a and b are not affected by the expression
// All integer, real, or decimal types
c = a + b

// lastName and firstName are unaffected

// All string variable types
employeeName = lastName + firstName

// All numerical variable types
myVariable = 12 * (yourVariable + ourVariable)
```

Precedence

Notice the parentheses in the expression example above. Parentheses play a role in precedence. Precedence is the order in which operations take place in a mathematical equation. This is

sometimes called *order of operation*. Any operation inside parentheses is evaluated first, and then followed by other operations.

If you evaluate the example:

```
myVariable = 12 * (yourVariable + ourVariable)
```

using 10 for *yourVariable* and 5 for *ourVariable*:

```
myVariable = 12 * (yourVariable + ourVariable)
```

the order of the operation is the following:

1. *yourVariable* is added to *myVariable* resulting in 15.
2. 15 is multiplied by 12 resulting in 180.
3. *myVariable* is assigned the value 180.

For further information on the different operators, please see Operators.

Conditional expressions

Conditional expressions assign a value to a variable depending on the result of an expression. The format of a conditional expression is as follows:

```
{Boolean expression} ? {expression} : {expression}
```

If the Boolean expression evaluates to *true*, the value to the left of

the colon is assigned. If it evaluates to *false*, the value to the right of the colon is assigned. Now, look the next example.

```
myResult = (show = 1) ? "on" : "off"
```

In this example, the variable *myResult* is assigned the value "on" if the value of the variable *show* is equal to 1. Otherwise, "off" is assigned to *myResult*.

Chapter 6. Operators

Operators

An operator is a symbol that performs a function on one, two or three operands. An operator that requires one operand is called a *unary* operator. If they require two or three operands, they are called *binary* or *ternary* operators.

Operators can be grouped in:

- Arithmetic operators that provide the ability to perform mathematical operations.
- Relational operators that provide the ability to compare two or more values.
- Logical operators that provide the ability to perform logic operations.
- Other operators that provide the ability to perform miscellaneous operations.

Arithmetic operators

The following table lists the operators that can be used in a method to perform mathematical calculations:

Fuego Style	Java Style	Visual Basic .Net Style	Meaning	Precedence
+	+	+	Addition	2
-	-	-	Subtraction	2
*	*	*	Multiplication	1
/	/	/	Division	1
rem	%	Mod	Remainder	1

Variables of Int, Decimal, Real, Time, and Interval types may be used as operands of arithmetic operators. The result is based on the following rules:

- If any of the operands is real, the result is real.
- If any of the operands is decimal and none are real, the result is decimal with sufficient places to hold the operation's sum, difference, multiplication, or remainder.
- If both operands are integer, the result is integer.
- Monadic '-' (change of sign) and '+' identity are supported.
- The result always has the highest precision of the operands.

Relational operators

Relational operations yield a Boolean result. Relational operators must be applied to homogenous variable types. For example, comparing an integer to another integer passes the check. However, comparing an integer to a Boolean results in an error. Any variable of numeric types is considered homogenous.

Logic operators

Logical operators can be used between Boolean variable types. The following table lists the operators that are available in Method:

Operator	Meaning	Order of Operation	Evaluation Criteria
not	Logical not	5	Negates the operand value so if the value is true, it becomes false and vice

Operator	Meaning	Order of Operation	Evaluation Criteria
			versa.
and	Logical and	6	Yields true if and only if both operands are true. Yields false if any of the operands are false. Uses evaluation by need, so if the first operand is false, the evaluation process is stopped and the result is false.
or	Logical or	7	Yields true if either operand is true. Yields false if and only if both operands are false. Uses evaluation by need, so if the first operand is true, the evaluation process is stopped and the result is true.

NOT Example:

```
if not found then
    // do something
end
```

AND Example:

```
if orderAmt < 200 and paymentType = "Credit" then
    // do something
end
```

OR Example:

```
if shipStatus = "Received" or shipStatus = "Pending" then
    // do something
end
```

Other operators

The 'is' operator is used to test whether a given variable belongs to a given type (or whether its value is null).

Syntax

```
<expression > is (<type-specification> | null)
```

Examples

The following example yields true if the value of 'a' is null:

```
if a is null then
  ...
end
```

The next example yields true if the value of 'a' is NOT null:

```
a is not null
```

This example displays 'It's a String' if the variable 'any' contains a String value:

```
any as Any
any = "this is a test"

if any is String then
  display "It's a String"
end
```

Chapter 7. Numbers

Numbers

FuegoBPM Studio supports three classes of numbers:

- Integer numbers
- Real numbers
- Decimal numbers

Integer numbers

Integer numbers are always signed and can be specified in three bases:

- Octal (e.g.: 0777)
- Decimal (e.g.: 511)
- Hexadecimal (e.g.: 0x1FF)

Octal numbers are prefixed by a 0 (zero). Hexadecimal numbers are prefixed by '0x'.

Integer numbers can also have a precision that limits the values that variables of this type can hold:

Precision: 64

Maximum value: 9223372036854775807

Minimum Value: -9223372036854775808

Precision: 32

Maximum value: 2147483647

Minimum Value: -2147483648

Precision: 16

Maximum value: 32767

Minimum Value: -32768

Precision: 8

Maximum value: 127

Minimum Value: -128

All other precision values are illegal.

Real numbers

Real numbers are implemented as floating point numbers according to IEEE 754 specification for floating point numbers.

All Real constants must include a '.' and they may have an exponential part denoted by 'e' or 'E' and are suffixed by an 'f', 'F', 'd' or 'D'. The suffix is optional if the exponent is included.

The following are all valid Real constants:

- 2.0f
- 2.0E20
- 5.324d

Real numbers can also have a precision that limits the values that variables of this type can hold:

Precision: 64

Maximum Value: 1.7976931348623157E308

Minimum Value: 4.9E-324

Precision: 32

Maximum Value: 3.4028235E38

Minimum Value: 1.4E-45

All other precision values are illegal.

Decimal numbers

Decimal numbers are arbitrary precision numbers. Each number has a precision and a scale. The precision specifies the number of decimal digits to the left of the decimal point. The scale is the number of digits to the right of the decimal point.

When you declare a decimal number you can do the following:

- Leave both floating.
- Fix the scale and leave the precision floating.
- Fix both, scale, and precision.

See the following example:

```
floating as Decimal  
fixedScale as Decimal(2)  
fixedBoth as Decimal(5, 2)
```

In the example above, *floating* is of arbitrary precision, *fixedScale* can be any number with only two digits to the right of the decimal digit (e.g.: 600000.25), and *fixedBoth* can hold only numbers with up to 5 decimal digits to the left of the decimal point and two to the right.

Decimal constants are a sequence of decimal digits followed by a dot ('.') followed by a sequence of decimal digits. Note that unlike Real constants, the number of digits to the right of the decimal point is significant and specifies the scale of the constant. Here is an example:

```
display 10.00 * 10.00
```

The code above displays "100.0000", a number with a scale of four (4). This is the minimum scale that can be guaranteed to hold the product of two numbers with scale two (2) without losing information.

Note that the scale affects how a number is displayed:

```
display 12.3456780 to Decimal(2)
```

The example above shows **12.34**.

Creating an Int in radix 10 from a String

```
intNumber as Int  
intNumber = Int("0001920")
```

The code above creates an Int number from a String based on decimal radix.

Creating a Int in a radix different than 10 from a Str

```
octal as Int  
  
octal = Int.parse(text : "2015", offset : 1, length : 3,  
                  radix : 8)
```

The code above creates an Octal.

- **text**: string containing the number to convert.
- **offset**: position of the string to use as beginning to convert the number.
- **length**: quantity of characters to take from the offset position.
- **radix**: number system to convert the string to.

In the example the substring *015* is converted into the octal number system. The **octal** variable will contain a *13*.

Real vs Decimal numbers

The choice between Real or Decimal numbers is important. Both can hold large numbers with a certain amount of precision, but there are fundamental differences between the two types of numbers:

- Real numbers are designed for speed in calculations where accuracy is not so important and where a close value is good enough.
- Decimal numbers are designed to provide a bound to the error you are willing to accept, sacrificing some performance if it is needed to guarantee the accuracy.

These differences become especially important when dealing with

money. As a rule, whenever you are handling numbers that represent money, use Decimal numbers.

Decimal Arithmetic

It is very important to bear in mind the rules that apply to decimal arithmetic when dealing with variables with different decimal precisions.

Here are the rules:

1. If you want a variable to handle a determined precision, you must declare it (*Decimal(precision)*).
2. In an addition or a subtraction, the result has the highest precision.
3. In a multiplication, the result has as precision the sum of the precisions of the two operands.
4. In a division, the result has the precision of the left operand (the *dividend*).
5. In an assignment,
 - a. it takes the resultant precision of the operation if the left operand has no precision defined or if its precision is higher.
 - b. it takes the left operand precision when it is lower than the operation resultant precision.
6. Every time the precision is reduced, it is *rounded*. That is, for example, a *0.5* is rounded to *1* and not to *0*.

Examples

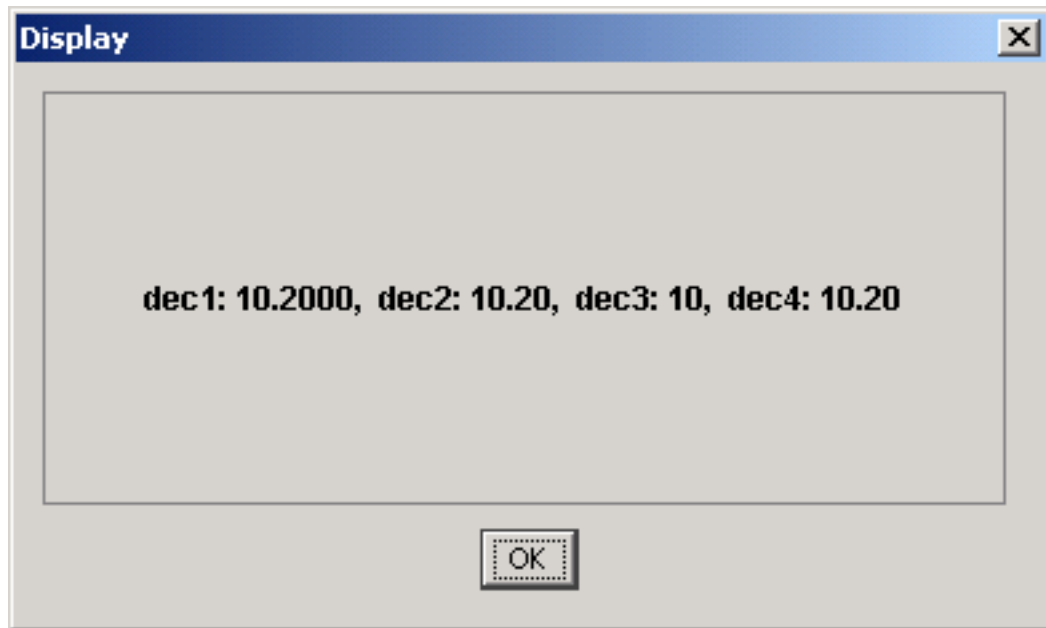
Some simple examples are given with the following variable declaration:

```
dec1 as Decimal(4)
dec2 as Decimal(2)
dec3 as Decimal(0)
dec4 as Decimal
res1 as Decimal
res2 as Decimal
res3 as Decimal
res4 as Decimal
```

Assignment

```
dec4 = 10.20
dec2 = dec4
dec1 = dec4
dec3 = dec4

display "dec1: " + dec1 + ", dec2: " + dec2 +
      ", dec3: " + dec3 + ", dec4: " + dec4
```

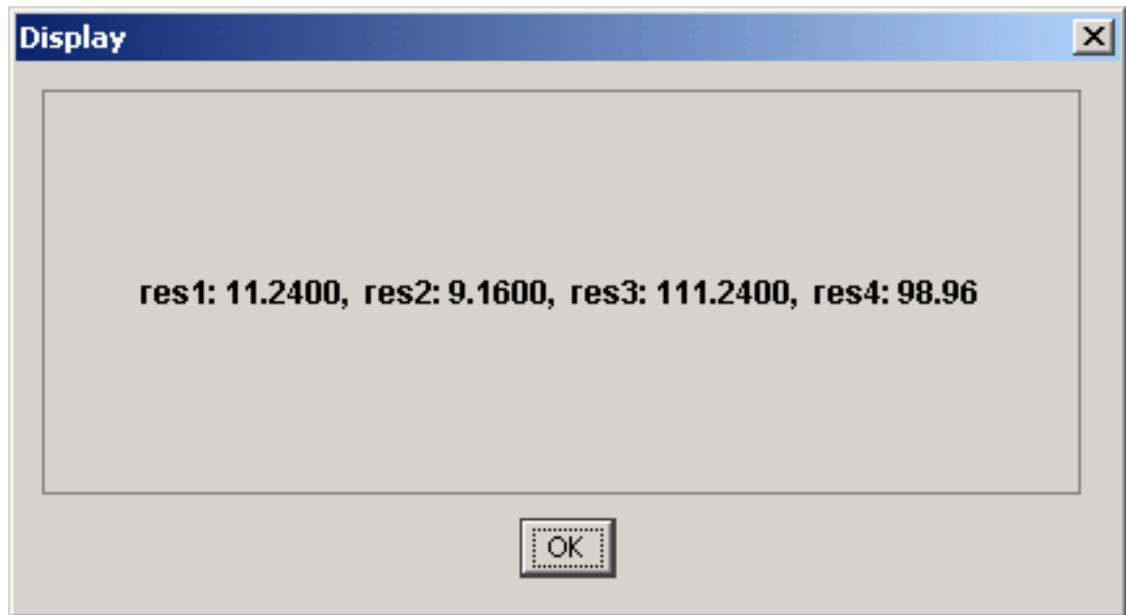


Addition

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003

res1 = dec1 + dec2
res2 = dec1 - dec2
res3 = dec1 + dec2 + dec3
res4 = dec3 - dec2

display "res1: " + res1 + ", res2: " + res2 +
      " , res3: " + res3 + ", res4: " + res4
```

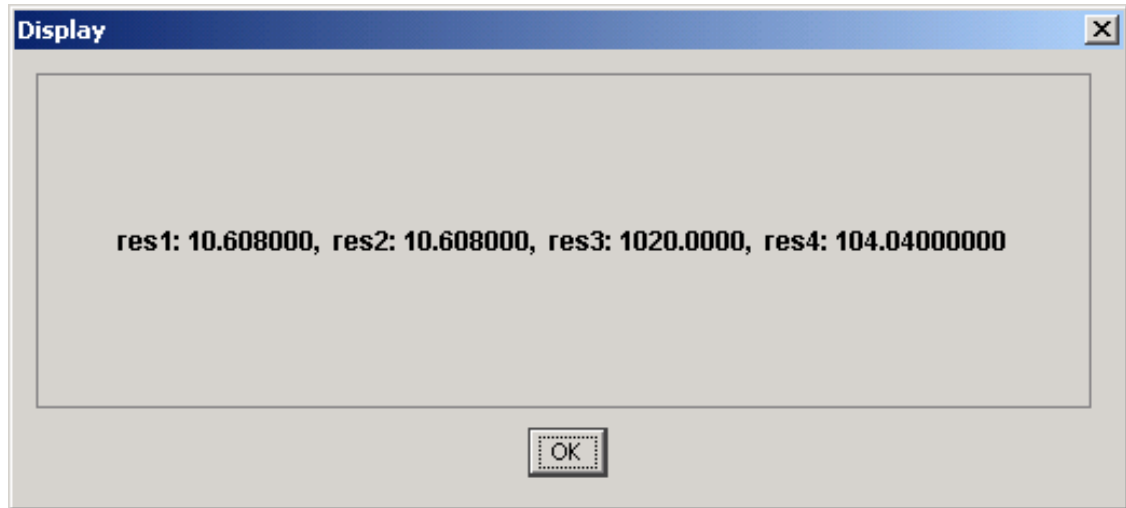


Multiplication

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003

res1 = dec1 * dec2
res2 = dec2 * dec1
res3 = dec3 * dec1
res4 = dec1 * dec1

display "res1: " + res1 + ", res2: " + res2 +
        ", res3: " + res3 + ", res4: " + res4
```

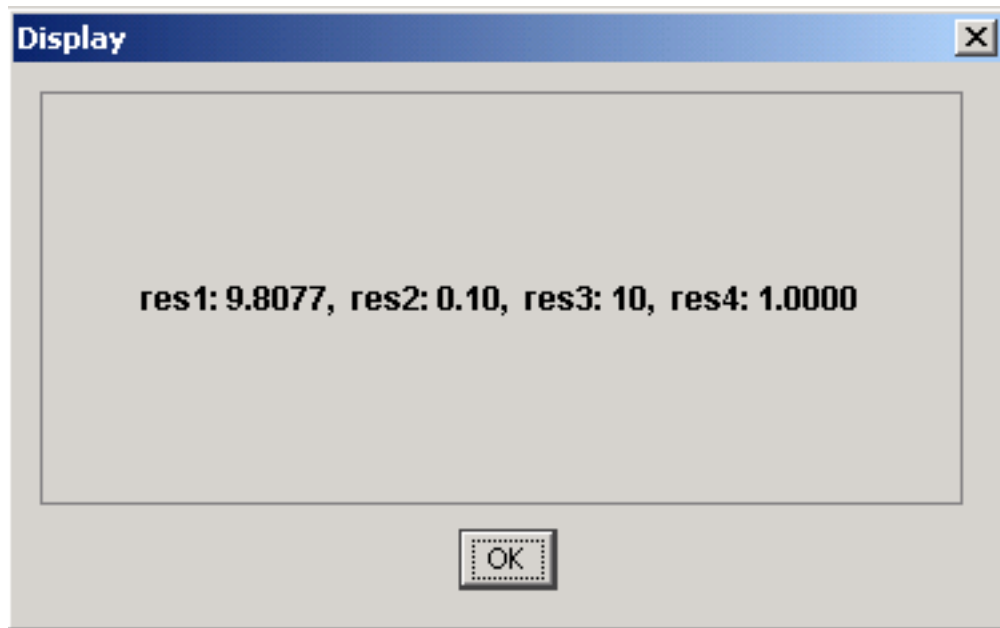



Division

```
dec1 = 10.20
dec2 = 1.04
dec3 = 100.003

res1 = dec1 / dec2
res2 = dec2 / dec1
res3 = dec3 / dec1
res4 = dec1 / dec1

display "res1: " + res1 + ", res2: " + res2 +
        ", res3: " + res3 + ", res4: " + res4
```



Number Functions

FuegoBPM implements methods to perform basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.

Technical Details

Real numbers are implemented as floating-point numbers. Therefore, the quality of implementation specifications concerns two properties: accuracy of the returned result and monotonicity of the method.

Accuracy of the floating-point methods is measured in terms of ulps, units in the last place. For a given floating-point format, an ulp of a specific real number value is the difference between the two floating-point values closest to that numerical value.

When discussing the accuracy of a method as a whole rather than at a specific argument, the number of ulps cited is for the worst-case error at any argument. If a method always has an error of less than 0.5 ulps, the method always returns the floating-point number

nearest to the exact result, such a method is correctly rounded. A correctly rounded method is generally the best a floating-point approximation can be. However, it is impractical for many floating-point methods to be correctly rounded. Instead, a larger error bound of 1 or 2 ulps is allowed for certain methods.

Informally, with 1 ulp error bound, the exact result should be returned when the exact result is a representative number. Otherwise, either of the two floating-point numbers closest to the exact result may be returned. In addition to accuracy in individual arguments, maintaining proper relations between the method and different arguments is also important. Therefore, methods with more than 0.5 ulp errors are required to be semi-monotonic; whenever the mathematical function is non-decreasing, so is the floating-point approximation. Likewise, whenever the mathematical function is non-increasing, so is the floating-point approximation. Not all approximations that have 1 ulp accuracy will automatically meet the monotonicity requirements.

Functions

abs

Returns the absolute value of a Real value. If the argument is not negative, the argument is returned. If the argument is negative, the negation of the argument is returned.

- Special cases (see the References in the footnotes):
 - If the argument is positive, zero, or negative zero, the result is positive zero.
 - If the argument is infinite, the result is positive infinity.
 - If the argument is NaN, the result is NaN.

Arguments

- Real - the argument whose absolute value is to be determined.

Returns

- Real - the absolute value of the argument.

acos

Returns the arc cosine of an angle, in the range of 0.0 through pi.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or its absolute value is greater than 1, the result is NaN.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - the value whose arc cosine is to be returned.

Returns

- Real - the arc tangent of the argument.

asin

Returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or its absolute value is greater than 1, the result is NaN.
 - If the argument is zero, the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - the value whose arc sine is to be returned.

Returns

- Real - the arc sine of the argument.

atan

Returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.

- Special cases (see the References in the footnotes):

- If the argument is NaN, the result is NaN.
- If the argument is zero, the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - the value whose arc tangent is to be returned.

Returns

- Real - the arc tangent of the argument.


ceil

Returns the smallest (closest to negative infinity) real value that is not less than the argument and is equal to a mathematical integer.

- Special cases (see the References in the footnotes):
 - If the argument value is already equal to a mathematical integer, the result is the same as the argument.
 - If the argument is positive zero or negative zero, the result is the same as the argument.
 - If the argument value is less than zero but greater than -1.0,

the result is negative zero.

Note

 The value of `ceil(Real)` is exactly the value of `-floor(-Real)`.

Arguments

- Real

Returns

- Real - the smallest (closest to negative infinity) floating-point value that is not less than the argument and is equal to a mathematical integer.

COS

Returns the trigonometric cosine of an angle.

- Special cases:
 - If the argument is NaN or an infinity, the result is NaN.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - an angle, in radians.

Returns

- Real - the cosine of the argument.

exp

Returns Euler's number e raised to the power of a real value.

- Special cases (see the References in the footnotes):
 - If the argument is NaN, the result is NaN.
 - If the argument is positive infinity, the result is positive infinity.
 - If the argument is negative infinity, the result is positive zero.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - the exponent to raise e to.

Returns

- Real - the value $e^{\text{exp } a}$, where e is the base of the natural

logarithms.

floor

Returns the largest (closest to positive infinity) real value that is not greater than the argument and is equal to a mathematical integer.

- Special cases (see the References in the footnotes):
 - If the argument value is already equal to a mathematical integer, the result is the same as the argument.
 - If the argument is NaN, an infinity, positive zero, or negative zero, the result is the same as the argument.

Syntax

Real = floor (Real)

Arguments

- Real - a value.

Returns

- Real - the largest (closest to positive infinity) floating-point value that is not greater than the argument and is equal to a mathematical integer.

log

Returns the natural logarithm (base e) of a real value.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or less than zero, the result is NaN.
 - If the argument is positive infinity, the result is positive infinity.
 - If the argument is positive zero or negative zero, the result is negative infinity.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - a number greater than 0.0.

Returns

- Real - the value \ln argument, the natural logarithm of argument.

max

Returns the greater of two Real values. That is, the result is the argument closer to positive infinity. If the arguments have the same value, the result is that same value. If either value is NaN, the result is NaN. Unlike the numerical comparison operators, this method considers negative zero to be strictly smaller than positive zero. If

one argument is positive zero and the other is negative zero, the result is positive zero (see the References in the footnotes).

Arguments

- Real - an argument.
- Real - another argument.

Returns

- Real - the larger of the two arguments.

min

Returns the smaller of two Real values. That is, the result is the argument closer to the value of Real.MIN_VALUE. If the arguments have the same value, the result is that same value.

Arguments

- Real- an argument.
- Real- another argument.

Returns

- Real- the smaller of the two arguments.

pow

Returns the value of the first argument raised to the power of the second argument.

- Special cases (see the References in the footnotes):
 - If the second argument is positive or negative zero, the result is 1.0.
 - If the second argument is 1.0, the result is the same as the first argument.
 - If the second argument is NaN, the result is NaN.
 - If the first argument is NaN and the second argument is nonzero, the result is NaN.
 - If
 - the absolute value of the first argument is greater than 1 and the second argument is positive infinity, or
 - the absolute value of the first argument is less than 1 and the second argument is negative infinity, the result is positive infinity.
 - If
 - the absolute value of the first argument is greater than 1 and the second argument is negative infinity, or
 - the absolute value of the first argument is less than 1 and the second argument is positive infinity, the result is positive zero.
 - If the absolute value of the first argument equals 1 and the second argument is infinite, the result is NaN.

- If
 - the first argument is positive zero and the second argument is greater than zero, or
 - the first argument is positive infinity and the second argument is less than zero, the result is positive zero.
- If
 - the first argument is positive zero and the second argument is less than zero, or
 - the first argument is positive infinity and the second argument is greater than zero, the result is positive infinity.
- If
 - the first argument is negative zero and the second argument is greater than zero but not a finite odd integer, or
 - the first argument is negative infinity and the second argument is less than zero but not a finite odd integer, the result is positive zero.
- If
 - the first argument is negative zero and the second argument is a positive finite odd integer, or
 - the first argument is negative infinity and the second argument is a negative finite odd integer, the result is negative zero.

- If
 - the first argument is negative zero and the second argument is less than zero but not a finite odd integer, or
 - the first argument is negative infinity and the second argument is greater than zero but not a finite odd integer, the result is positive infinity.
- If
 - the first argument is negative zero and the second argument is a negative finite odd integer, or
 - the first argument is negative infinity and the second argument is a positive finite odd integer, the result is negative infinity.
- If the first argument is finite and less than zero
 - if the second argument is a finite even integer, the result is equal to the result of raising the absolute value of the first argument to the power of the second argument.
 - if the second argument is a finite odd integer, the result is equal to the negative of the result of raising the absolute value of the first argument to the power of the second argument
 - if the second argument is finite and not an integer, the result is NaN.
- If both arguments are integers, the result is exactly equal to the mathematical result of raising the first argument to the

power of the second argument if that result can, in fact, be represented exactly as a real value.

(In the foregoing descriptions, a floating-point value is considered to be an integer if and only if it is finite and a fixed point of the method `ceil` or, equivalently, a fixed point of the method `floor`. A value is a fixed point of a one-argument method if and only if the result of applying the method to the value is equal to the value.)

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - the base.
- Real - the exponent.

Returns

- Real

round

Returns the closest int to the argument. The result is rounded to a real by adding $1/2$, taking the floor of the result and casting the result to type int. In other words, the result is equal to the value of the expression:

$\text{Real} = \text{floor}(\text{Real} + 0.5f)$

- Special cases (see the References in the footnotes):
 - If the argument is NaN, the result is 0.
 - If the argument is negative infinity or any value less than or equal to the value of Real.MIN_VALUE, the result is equal to the value of Real.MIN_VALUE.
 - If the argument is positive infinity or any value greater than or equal to the value of Real.MAX_VALUE, the result is equal to the value of Real.MAX_VALUE.

Arguments

- Real - a floating-point value to be rounded.

Returns

- Real - the value of the argument rounded to the nearest int value.

sin

Returns the trigonometric sine of an angle.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or an infinity, the result is NaN.
 - If the argument is zero, the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - an angle, in radians.

Returns

- Real - the sine of the argument.

sqrt

Returns the correctly rounded positive square root of a Real value.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or less than zero, the result is NaN.
 - If the argument is positive infinity, the result is positive infinity.
 - If the argument is positive zero or negative zero, the result is the same as the argument.

Otherwise, the result is the real value closest to the true mathematical square root of the argument value.

Arguments

- Real - a value.

Returns

- Real - the positive square root of argument. If the argument is NaN or less than zero, the result is NaN.

tan

Returns the trigonometric tangent of an angle.

- Special cases (see the References in the footnotes):
 - If the argument is NaN or an infinity, the result is NaN.
 - If the argument is zero, the result is a zero with the same sign as the argument.

A result must be within 1 ulp of the correctly rounded result. Results must be semi-monotonic.

Arguments

- Real - an angle, in radians.

Returns

- Real - the tangent of the argument.

Chapter 8. Strings

Strings

A string is zero or more characters put together. A character is anything that you can type, such as letters, digits, punctuation, and spaces. A string is useful to hold any text.

String literals

A string literal consists of zero or more characters enclosed in double quotes. Each character may be represented by an escape sequence.

The following are examples of string literals:

```
// the empty string
display ""
// a string containing 16 characters
display "This is a string"
// a string containing double-quote alone
display "\\\""
```

In Fuego style, consecutive strings are automatically merged:

```
display "This is a string "
      "Made from separate strings"
```

The above code displays the string, "This is a string Made from separate strings".

Escape sequences

The character and string escape sequences allow for the representation of some nongraphic characters as well as the single quote, double quote, and backslash characters in string literals.

```
// \\u0007: bell BEL
display "\\a"
// \\u000B: vertical tab VT
display "\\v"
// \\u0008: backspace BS
display "\\b"
// \\u0009: horizontal tab HT
display "\\t"
// \\u000a: linefeed LF
display "\\n"
// \\u000c: form feed FF
display "\\f"
// \\u000d: carriage return CR
display "\\r"
// \\u0027: single quote '
display "\\'"
// \\u005c: backslash \
display "\\\"
// \\u0022: double quote
display "\\\""
```

Other escape sequences can be written in hexadecimal, prefixing a four-digit hexadecimal number with "\\u" ("u" stands for unicode).

String concatenation

Strings can be concatenated at runtime by using the string concatenation operator '+'. You can also use any type to build a string since at least one of the components is a string:

```
total as Int
total = 200
display "Total is: " + total + " units"
```

Regular Expressions

Advanced string manipulation is done through regular expressions. For further information, please see Regular Expressions.

String Functions

substring

Returns a new string that is a substring of this string. The substring begins with the character at the specified index and extends to the end of this string.

Arguments

- String: the string on which the function operates.
- Int first: the beginning index, inclusive.

Returns

- String: the specified substring.

Example

```
text as String
text = "Hello World"
display substring(text, first : 5)
```

The previous example displays "World!".

substring

Returns a new string that is a substring of this string. The substring begins at the specified *first* index and extends to the character at index *last* - 1. Thus, the length of the substring is *last-first*.

Arguments

- String: the string on which the function operates.
- Int first: the beginning index, inclusive.
- Int last: the ending index, exclusive.

Returns

- String: the specified substring.

Example

```
text as String
text = "Hello World!"
display substring(text, first : 5, last : 11)
```

The previous example displays "World".

fields

Given a source string and delimiter character, it returns an array of strings containing substrings of the original string delimited by the specified character. The delimiter is not included in the result.

Arguments

- String: the string on which the function operates.
- String delim: Character that delimits field.

Returns

- `String[]`: an array of strings containing the fields delimited by *delim*.

Example

```
text as String
text = "Hello World!"
display fields(text, delim : " ")
```

The example above displays ["Hello", "World!"].

length

Returns the number of characters in the string.

Arguments

- `String`: the string on which the function operates.

Returns

- `Int`: Number of characters in the string.

Example

```
text as String
text = "Hello World!"
display length(text)
```

The previous example displays 12.

replace

Returns a new string with all the occurrences of *from* in the original string replaced by *to*.

This function has a variation that accepts a regular expression for matching the pattern to be replaced. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.
- String from: the string to find.
- String to: the replacement text.

Returns

- String: a new string with the replacements.

Example

```
text as String
text = "Hello World!"
display replace(text, from : "World", @to : "Mary")
```

The previous example displays "Hello Mary!".

charAt

This function returns the character contained in the specified index position.

Arguments

- String: the string on which the function operates.
- Int position: zero-based index of a character inside the string.

Returns

- String(1): the character at the specified index.

Example

```
text as String
text = "Hello World!"
display charAt(text, position : 6)
```

The previous example displays "W".

indexOf

Searches inside a string for another string and returns the index where the first occurrence happens.

This function has a variation that accepts a regular expression for matching. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.
- String text: the text to find.

Returns

- Int: the index of the occurrence or -1 if not found.

Example

```
text as String
text = "Hello World!"
display indexOf(text, text : "Wor")
```

The previous example displays 6.

lastIndexOf

Searches inside a string for another string and returns the index where the last occurrence happens.

This function has a variation that accepts a regular expression for matching. See Regular Expressions for details.

Arguments

- String: the string on which the function operates.
- String text: the text to find.

Returns

- Int: the index of the occurrence or -1 if not found.

Example

```
text as String
text = "Hello World!"
display lastIndexOf(text, text : "o")
```

The previous example displays 7.

split

Splits a string using a regular expression. The delimiters are not included. See Regular Expressions for details.

Example

```
text as String= "One Two Three"
display split(text, '/\w+ \w+/m')
```

The previous example produces this output ["", "Three"].

count

This function counts the number of times that a character is found in a string.

Arguments

- String: the string on which the function operates.
- String(1) ch: character to find.

Returns

- Int: the number of occurrences of the specified character in the string.

Example

```
date as String = "10/12/2004"
if count (date, ch: "/") = 2 then
    date = replace(date, "/", "-")
    display date
else
    display "Bad Date Format"
end
```

The previous example displays "10-12-2004".

toUpperCase

Returns a new string with all the characters in uppercase.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the characters in uppercase.

Example

```
text as String
text = "Hello World!"
display toUpperCase(text)
```

The previous example displays "HELLO WORLD!".

toLowerCase

Returns a new string with all the characters in lowercase.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the characters in lowercase.

Example

```
text as String
text = "Hello World!"
display toLowerCase(text)
```

The previous example displays "hello world!".

trim

Returns a new string with all the whitespace removed from the beginning and the end of the string.

Arguments

- String: the string on which the function operates.

Returns

- String: a new string with all the leading a trailing whitespace removed.

Example

```
option as String = " Yes  "
if toLowerCase(trim (option))= "yes" then
  display "The option is correct"
else
  display "The option is wrong".
end
```

isMatch

Checks whether a string matches a regular expression. See Regular Expressions for details.

Example

```
text as String= "One Two Three"
display isMatch(text, '/\w+ Two \w+/g')
```

The previous example displays true.

contains

This function returns true if a substring in a text matches the

specified regular expression. See Regular Expressions for details.

Example

```
text as String= "One Two Three Four Five"
display contains(text, '/\w+ Tw/g')
```

The previous example displays true.

chars

Returns an array of String(1) containing all the characters in the string.

Arguments

- String: the string on which the function operates.

Returns

- String(1)[]: the characters in the string.

Example

```
text as String= "fuego"
characters as String(1)[]
characters = chars(text)
for each i in characters
do
    display "Char is " + i
end
```

The previous example displays:

```
"Char is f"  
"Char is u"  
"Char is e"  
"Char is g"  
"Char is o"
```

pad

Returns a new string completed with spaces until the specified length is reached.

Arguments

- String: the string on which the function operates.
- Int len: the desired length of the string. If you pass a negative number (e.g.: -1), it is ignored and returns an empty string.

Returns

- String: a new string of the specified length.

Example

```
text as String  
text = "Hello World!"  
display pad(text, len : 20)
```


strip

Returns a new string truncated to a specified length. If the string is shorter than *len*, it is left as it is.

Arguments

- String: the string on which the function operates.
- Int len: the desired length of the string. If you pass a negative number (e.g.: -1), it is ignored and returns an empty string.

Returns

- String: a new string of the specified length.

Example

```
text as String
text = "Hello World!"
display strip(text, len : 5)
```

The previous example displays "Hello".

How to convert a String to a Time

Example

```
strDate      = "Tue Feb 22 15:26:02 ART 2005"
strPattern   = "EEE MMM dd HH:mm:ss z yyyy"

simpleDateFormat = Java.Text.SimpleDateFormat(strPattern)
realDate = parse(simpleDateFormat, strDate)
```

```
display realDate
```

realDate 's value becomes **2005-02-22 15:26:02-03**

realDate is a variable of *time* type. It contains the same date that was expressed as a string in **strDate**.

String Attributes

Empty

This attribute returns true if the string is empty (its length is zero).

Returns

- Boolean: true if its length is zero. False otherwise.

Example

```
text as String  
text = "Hello World!"  
display text.empty
```

Chapter 9. Times and Intervals

Times and Intervals

FuegoBPM Studio supports two built-in types for time management:

- Time - represents a specific point in time.
- Interval - represents the difference between two moments in time.

Times are stored as the number of microseconds since Jan 1st, 1970 (also known as UNIX epoch).

Intervals are stored as months, days, and microseconds.

Times

The string representation of a time is ISO 8601 compliant. When converting a string into a time, FuegoBPM Studio supports a somewhat relaxed subset of ISO 8601; even dates without separators are accepted. The accepted string formats match those of time literals.

Time Literals

Time literals are specified between single quotes. The following is a list of valid time literals:

```
'23:30'  
'23:30:23'  
'23:30:23.001023'  
'23:30:23.001023Z'  
'23:30:23.001023-05'  
'23:30:23.001023-3:30'
```

```
'1995-02-03'  
'1995-02-03 23:30'  
'1995-02-03 23:30:23'  
'1995-02-03 23:30:23.001023'  
'1995-02-03 23:30:23.001023Z'  
'1995-02-03 23:30:23.001023-05'  
'1995-02-03 23:30:23.001023-3:30'  
'1995-02-03T23:30'  
'1995-02-03T23:30:23'  
'1995-02-03T23:30:23.001023'  
'1995-02-03T23:30:23.001023Z'  
'1995-02-03T23:30:23.001023-05'  
'1995-02-03T23:30:23.001023-3:30'  
'19950203T'  
'19950203T233023.001023-330'
```

Time Zones

Following ISO 8601, time zones are specified as offsets from UTC. Named time zones are not supported because there is no international standard for time zone abbreviations. If no time zone is specified in a time literal, the default time zone for the current locale is used.

Note that:

- Interactive methods run in the locale of the current user.
- Automatic methods run in the FuegoBPM Engine's locale.

When a time is presented to a user, the format associated to the user's locale is used. For custom time formatting, the `format()` function can be used. For further information, please see [Time Functions](#).

Intervals

Intervals have two primary parts:

- Two calendar dependent components (months and days)
- A calendar independent component (hours, minutes, seconds and microseconds)

The calendar dependent component exists, so arithmetic between time and interval is consistent. When using a Gregorian calendar (the most common calendar in use), you cannot assume that a month equals to 30 days. In fact, you cannot even assume that a day lasts 24 hours.

The Gregorian calendar inserts two corrections:

- The leap year - Every four years a day is added to February, unless the year is a multiple of 100 and not a multiple of 400 (that is why the year 2000 had 366 days, instead of 365 like 1900).
- The leap second - Sometimes a second is added or removed from the last minute of certain days to cope with the accumulated error caused by the earth's change of speed.

So, for example, if you want to obtain a time two months from now, you can do:

```
display 'now' + '2M'
```

Interval Literals

Interval literals are enclosed by single-quotes ('). They are formed

by a sequence of fields, where each field is a number plus a unit suffix. The following table lists all valid suffixes:

Unit Suffix	Description
Y	Years
M	Months
d or D	Days
h or H	Hours
m	Minutes
s or S	Seconds
x	Microseconds

Note



A 'T' in the constant forces the interpretation of 'M' to be equal to 'm', e.g.: '2MT2M' equals '2M2m'.

All magnitudes can contain a '.' to express a fractional part, although the fractional part is dropped for days or months.

The following example shows some valid Interval constants:

```
display '2MT2.5M'
display '1Y1M3h2m1.500s'
display '1.5h'
```

Arithmetic

As mentioned before, time and intervals have some arithmetic rules.

The following table lists the behavior of addition and subtraction with time and interval:

Operations	Result Type
Time - Time	Interval
Time + Interval (or Interval + Time)	Time
Time - Interval	Time
Interval + Interval	Interval
Interval - Interval	Interval

For further information on Time and Interval, please refer to the following:

- Time Attributes
- Interval Attributes
- Time and Interval Functions

Time Attributes

A Time object contains several attributes to manipulate the different components of time, such as days and hours. The following table lists all Time's attributes and provides some examples:

Attribute	Description
AD	Field that indicates the calendar era indicating the common era (Anno Domini.).

Example:

```
time as Time
/*this will display 1 because
the current era is AD*/
```

```
display time + "\n\n AD = " + time.AD
```

Attribute	Description
am	Boolean value which indicates whether the Time's period is between midnight to just before noon.

Example:

```
time as Time
time = '2004-12-25 02:45:00-03'
//this will display true
display time + "\n\n am = " + time.am
time = '2004-12-25 20:45:00-03'
display time + "\n\n am = " + time.am
```

Attribute	Description
ampm	Field that indicates the period of the day. If the period is am, it will return 0; otherwise, it will return 1.

Example:

```
time as Time
time = '2004-12-25 02:45:00-03'
//this will display 0
display time + "\n\n ampm = " + time.ampm
time = '2004-12-25 20:45:00-03'
//this will display 1
display time + "\n\n ampm = " + time.ampm
```


Attribute	Description
BC	Field that indicates the calendar era indicating the period before the common era (before Christ).

Example:

```
time as Time
/*this will display 0 because
the current era is not BC*/
display time + "\n\n BC = " + time.BC
```

Attribute	Description
date	String value containing the date formatted with the default mask.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
//this will display Dec 25, 2004
display time + "\n\n" + time.date
```

Attribute	Description
day	Int value representing the day component of the time.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
```

```
//it will display 25
display time + "\n\n day: " + time.day
```

Attribute	Description
dayOfMonth	Int value representing the day part of the time.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
//this will display 25
display time + "\n\n day of month: " +
time.day
```

Attribute	Description
dayOfWeek	Day of the week. Returns an integer from 1 to 7, where Sunday's position is 1.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n day of the week: " +
time.dayOfWeek
```

Attribute	Description
days	Returns the amount of days passed since 00:00 January 1st, 1970 GMT (UNIX epoch).

Example:

```
time as Time
display "days since EPOCH: " + time.days
```

Attribute	Description
EPOCH	1969-12-31 00:00:00-00. Base time from which milliseconds to calculate dates are counted.

Example:

```
//1969-12-31 00:00:00-00
display "EPOCH: " + Time.EPOCH
```

Attribute	Description
firstDayOfMonth	Time value corresponding to the first day of the month.

Example:

```
time as Time
//firstDayOfMonth is a Time object
display "first day of month: " +
time.firstDayOfMonth
```

Attribute	Description
hour	Int value representing the hour component of the date in the format h.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n hour: " + time.hour
```

Attribute	Description
hourOfDay	Int value representing the hour component of the date in the format hh.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n hour of the day: " +
time.hour
```

Attribute	Description
hours	Returns the amount of hours passed since January 1st, 1970 GMT.

Example:

```
time as Time
display "hours passed since EPOCH: "+
time.hours
```

Attribute	Description
lastDayOfMonth	Time value corresponding to the last day of the month.

Example:

```
time as Time
//lastDayOfMonth is a Time object
display "last day of month: " +
time.lastDayOfMonth
```

Attribute	Description
locale	This attribute is used to change the current locale. It is a write only attribute.

Example:

```
time as Time
display "date fomatted with default locale: " +
time.formatDate
Time.locale = Java.Util.Locale.GERMAN
display "date with German locale: " +
time.formatDate
```

Attribute	Description
maxvalue	The maximum value a Time object can have.

Example:

```
display "Time object's maximum value: "
+ Time.maxvalue
```

Attribute	Description
microSecond	Int value representing the microseconds component of the date.

Example:

```
time as Time
display time + "\n\n microseconds in time: " +
time.microSecond
```

Attribute	Description
microSeconds	Returns the amount of microseconds passed since January 1st, 1970 GMT.

Example:

```
time as Time
display "microseconds since EPOCH: " +
time.microDeconds
```

Attribute	Description
milliSeconds	Returns the amount of milliseconds passed since January 1st, 1970 GMT.

Example:

```
time as Time
display "milliseconds since EPOCH: " +
```

```
time.milliseconds
```

Attribute	Description
minute	Int value representing the minutes component of the date.

Example:

```
time as Time
time = '2004-12-25 20:45:00-03'
display time + "\n\n minutes: " + time.minute
```

Attribute	Description
minutes	Returns the amount of minutes passed since January 1st, 1970 GMT.

Example:

```
time as Time
display "minutes since EPOCH: " + time.minutes
```

Attribute	Description
minvalue	The minimum value a Time object can have.

Example:

```
time as Time
```

```
display "Time object's minimum value: "  
+ time.minvalue
```

Attribute	Description
month	Int value representing the month component of the date.

Example:

```
time as Time  
time = '2004-12-25 20:45:00-03'  
display time + "\n\n month: " +  
time.month
```

Attribute	Description
second	Int value representing the seconds component of the date.

Example:

```
time as Time  
time = '2004-12-25 20:45:10-03'  
display time + "\n\n seconds: " + time.second
```

Attribute	Description
seconds	Returns the amount of seconds passed since January 1st, 1970 GMT.

Example:


```
time as Time
display "seconds since EPOCH: " + time.seconds
```

Attribute	Description
timeOnlyHour	Int value representing the hours component of the time without calendar corrections.

Example:

```
time as Time
display time + "\n\n" +
"hours: " +
time.timeOnlyHour
```

Attribute	Description
timeOnlyMicroSecond	Int value representing the microseconds component of the time without calendar corrections.

Example:

```
time as Time
display time + "\n\n" +
"microseconds: "+
time.timeOnlyMicroSecond
```

Attribute	Description
timeOnlyMinute	Int value representing the minutes component of the time without calendar corrections.

Example:

```
time as Time
display time + "\n\n"+
"minutes: " +
time.timeOnlyMinute
```

Attribute	Description
timeOnlySecond	Int value representing the seconds component of the time without calendar corrections.

Example:

```
time as Time
display time + "\n\n"+
"seconds: " +
time.timeOnlySecond
```

Attribute	Description
time	String value containing the time of the day formatted with the default mask.

Example:

```
time as Time
display "time formatted with default mask:" +
time.date
```

Attribute	Description
timeZone	Time according to the locale.

Example:

```
time as Time
Time.timeZone = TimeZone.getTimeZone("GMT-3")
display "GMT-3: " + time
Time.timeZone = TimeZone.getTimeZone("GMT-8")
display "GMT-8: " + time
```

Attribute	Description
weekOfMonth	Int value indicating the week number within the current month, starting from 1.

Example:

```
time as Time
time = '2004-01-01 20:45:00-03'
display time + "\n\n week of month: " +
time.weekOfMonth
time = '2004-01-07 20:45:00-03'
display time + "\n\n week of month: " +
time.weekOfMonth
```

Attribute	Description
weekOfYear	Int value indicanting the week number within the current year, starting from 1.

Example:

```
time as Time
time = '2004-01-07 20:45:00-03'
display time + "\n\n week of year: " +
time.weekOfYear
time = '2004-02-07 20:45:00-03'
```

```
display time + "\n\n week of year: " +  
time.weekOfYear
```

Interval Attributes

Attribute	Description
ONE_DAY	Interval value representing an interval of one day.

Example:

```
interval as Interval  
interval = '25d5h1m'  
display "original interval: " + interval  
interval = interval + Interval.ONE_DAY  
display "after adding a day: " + interval
```

Attribute	Description
ONE_HOUR	Interval value representing an interval of one hour.

Example:

```
interval as Interval  
interval = '25d5h1m'  
display "original interval: " + interval  
interval = interval + Interval.ONE_HOUR  
display "after adding an hour: " + interval
```

Attribute	Description
ONE_MINUTE	Interval value representing an

Attribute	Description
	interval of one minute.

Example:

```
interval as Interval
interval = '25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_MINUTE
display "after adding a minute: " + interval
```

Attribute	Description
ONE_MONTH	Interval value representing an interval of one month.

Example:

```
interval as Interval
interval = '2M25d5h1m'
display "original interval: " + interval
interval = interval + Interval.ONE_MONTH
display "after adding a month: " + interval
```

Attribute	Description
ONE_SECOND	Interval value representing an interval of one second.

Example:

```
interval as Interval
interval = '25d5h1m7s'
display "original interval: " + interval
interval = interval + Interval.ONE_SECOND
display "after adding a second: " + interval
```

--

Attribute	Description
ZERO	Interval representing the zero value. When added to another interval, the interval does not change its value.

Example:

```
interval as Interval
interval = '1M20d3h40m5s'
display "original interval: " + interval
interval = interval + Interval.ZERO
display "after adding ZERO: " + interval
```

Attribute	Description
daysOnly	Int value representing the days component of an interval.

Example:

```
interval as Interval
interval = '1M20d10h'
display "days component of an interval: " +
interval.daysOnly
```

Attribute	Description
days	Int value representing the days component of an interval.

Example:

```
interval as Interval
interval = '1M20d10h'
display "days component of an interval: " +
interval.days
```

Attribute	Description
hoursOnly	Int value representing the hours component of an interval.

Example:

```
interval as Interval
interval = '1d20h30m'
display "hours component of an interval: " +
interval.hoursOnly
```

Attribute	Description
hours	Int value representing the hours component of an interval.

Example:

```
interval as Interval
interval = '1d20h30m'
display "hours component of an interval: " +
interval.hours
```

Attribute	Description
microSecondsOnly	Int value representing the microseconds component of an interval.

Example:

```
interval as Interval
interval = Interval("20.000003s")
display "microseconds component of an interval: " +
interval.microSecondsOnly
```

Attribute	Description
microSeconds	Total amount of microseconds contained in the interval.

Example:

```
interval as Interval
interval = Interval("20.000003s")
display "total microseconds of an interval: " +
interval.microSeconds
```

Attribute	Description
milliSecondsOnly	Int value representing the milliseconds component of an interval, without including microseconds.

Example:

```
interval as Interval
interval = '20.250320s'
display "milliseconds component of an interval: " +
interval.milliSecondsOnly
```


Attribute	Description
milliSeconds	Int value representing the milliseconds component of an interval plus the microseconds contained in it.

Example:

```
interval = '20.250320s'
display "milliseconds component with microseconds: " +
interval.milliSeconds
```

Attribute	Description
minutesOnly	Int value representing the minutes component of an interval.

Examples:

```
interval as Interval
interval = '2h20m10s'
display "minutes component of an interval: " +
interval.minutesOnly
```

Attribute	Description
minutes	Total amount of minutes contained in the interval.

Examples:

```
interval as Interval
interval = '2h20m10s'
```

```
display "total minutes of an interval: "+
interval.minutes
```

Attribute	Description
monthsOnly	Int value representing the months component of an interval.

Examples:

```
interval as Interval
interval = '1Y2M3d20h'
display "months component of an interval: " +
interval.monthsOnly
```

Attribute	Description
months	Total amount of months in an interval.

Examples:

```
interval as Interval
interval = '1Y2M3d20h'
display "total months of an interval: " +
interval.months
```

Attribute	Description
secondsOnly	Int value representing the seconds component of an interval.

Examples:

```
interval as Interval
interval = '1h20m35s'
display "seconds component of an interval: " +
interval.secondsOnly
```

Attribute	Description
seconds	Total amount of seconds in an interval.

Examples:

```
interval as Interval
interval = '1h1m5s'
display "total seconds of an interval " +
interval.seconds
```

Attribute	Description
totalMicroSeconds	Int value representing the total amount of microseconds contained in the interval.

Examples:

```
interval as Interval
interval = '20.250320s'
display "total microseconds of an interval: " +
interval.totalMicroseconds
```

Attribute	Description
yearsOnly	Int value representing the years component of an interval.

Examples:

```
interval as Interval
interval = '2Y10M15d'
display "years component of an interval: " +
  interval.yearsOnly
```

Attribute	Description
years	Total amount of years contained in the interval.

Examples:

```
interval as Interval
interval = '25M15d'
display "total years of an interval: " +
  interval.years
```

Time Functions

addDays

Adds a specified amount of days to a Time object.

Arguments

- Time - the Time object to which days will be added.
- Int i - the amount of days to be added to the time object.

Returns

The Time object resulting from adding the specified amount of days

to the given time.

Example

The following example adds 15 days to the current time and displays the result.

```
display "time in 15 days will be: " +  
    addDays('now', i : 15)
```

addHours

Adds a specified amount of hours to a Time object.

Arguments

- Time - the Time object to which hours will be added.
- Int i - the amount of hours to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of hours to the given time.

Example

The following example adds 12 hours to the current time and displays the result.

```
display "time in 12 hours will be: " +  
    addHours('now', i : 12)
```

addMicroSeconds

Adds a specified amount of microseconds to a Time object.

Arguments

- Time - the Time object to which microseconds will be added.
- Int i - the amount of microseconds to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of microseconds to the given time.

Example

The following example adds 500 microseconds to the current time and displays the result.

```
display "time in 500 microseconds will be: " +  
    addMicroSeconds('now', i : 500)
```

addMilliSeconds

Adds a specified amount of milliseconds to a Time object.

Arguments

- Time - the Time object to which milliseconds will be added.
- Int i - the amount of milliseconds to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of milliseconds to the given time.

Example

The following example adds 50,000 milliseconds to the current time and displays the result.

```
display "time in 50000 milliseconds will be: " +  
    addMilliseconds('now', i : 50000)
```

addMinutes

Adds a specified amount of minutes to a Time object.

Arguments

- Time - the Time object to which the milliseconds will be added.
- Int i - the amount of minutes to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of milliseconds to the given time.

Example

The following example adds 30 minutes to the current time and displays the result.

```
display "time in 30 minutes will be: " +  
    addMinutes('now', i : 30)
```

addMonths

Adds a specified amount of months to a Time object.

Arguments

- Time - the Time object to which the months will be added.
- Int i - the amount of months to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of months to the given time.

Example

The following example adds 6 months to the current time and displays the result.

```
display "time in 6 months will be: " +  
    addMonths('now', i : 6)
```

addSeconds

Adds a specified amount of seconds to a Time object.

Arguments

- Time - the Time object to which the seconds will be added.
- Int i - the amount of seconds to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of seconds to the given time.

Example

The following example adds 50 seconds to the time object and displays the result.

```
display "time in 50 seconds will be: " +  
    addSeconds('now', i : 50)
```

addWeeks

Adds a specified amount of weeks to a Time object.

Arguments

- Time - the Time object to which the weeks will be added.
- Int i - the amount of weeks to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of weeks to the given time.

Example

The following example adds 50 weeks to the current time and displays the result.

```
display "time in 50 weeks will be: " +  
    addWeeks('now', weeks : 50)
```

addYears

Adds a specified amount of years to a Time object.

Arguments

- Time - the Time object to which the years will be added.
- Int i - the amount of years to be added to the Time object.

Returns

The Time object resulting from adding the specified amount of years to the given time.

```
display "time in 10 years will be: " +  
    addYears('now', i : 10)
```

add

Adds a specified interval of time to a Time object.

Arguments

- Time - the Time object to which the interval of time will be added.
- Interval i - interval of time to add to the Time object.

Returns

The Time object resulting from adding the specified interval to the given time.

Example

The following example adds 5 days, 15 hours, and 30 minutes to the current time and displays the result.

```
display "time in 5 days 15 hours and 30 minutes: \n\n" +  
  add('now', interval : '5d15h30m')
```

between

Determines if the given Time object is between two Time objects.

Arguments:

- Time - the given Time object.
- Time from - the upper bound of the time period in which to search the given time, exclusive.
- Time to - the lower bound of the time period in which to search the given time, inclusive.

Returns

true - if the given Time object is contained in the specified period.

false - if the given object is not contained in the specified period.

Example

The following example finds out if the current time is in between the first day and last day of year 2004.

```
display "is today between 2004-01-01 and 2004-12-31? "+  
  between('now', from : '2004-01-01 12:00:00',  
  @to : '2004-12-31 12:00:00')
```

daysSince

Calculates the days passed between a given time and another time.

Arguments

- Time - the Time object.
- Time t - the other Time object.

Returns

An Int value representing the number of days between the two given times.

Example

The following example defines a Time variable birthdate, calculates the days passed between 'now' and birthdate and displays the result.

```
birthdate as Time
birthdate = '1979-02-19'
display "days passed since birthdate: " +
    daysSince('now', t : birthdate)
```

formatDate

Formats the Time object with the default mask.

Arguments

- Time - the Time object to be formatted.

Returns

A String containing the representation of the Time object formatted with the default mask.

Example

The following example displays the current time formatted with the default mask.

```
display formatDate('now')
```

formatDate

Formats the Time object with the specified date formatting style for the default locale.

Arguments

- Time - the Time object to be formatted.
- Int style - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the representation of the Time object formatted with the specified style.

Example

The following example displays the current time with the four possible formatting styles.

```
defaultDate as String = "Default format --> " +  
    formatDate('now', dateStyle : Time.DEFAULT)
```

```
fullDate as String = "Full format --> " +  
    formatDate('now', dateStyle : Time.FULL)  
  
longDate as String = "Long format --> " +  
    formatDate('now', dateStyle : Time.LONG)  
  
shortDate as String = "Short format --> " +  
    formatDate('now', dateStyle : Time.SHORT)  
  
display defaultDate + "\n\n" + fullDate + "\n\n" +  
    longDate + "\n\n" + shortDate + "\n\n"
```

formatTimeOnly

Formats this Time object as a time only, with no time zone correction.

Arguments

- Time - The Time object to be formatted.

Returns

A String containing the representation of the Time object formatted as time only.

Example

The following example displays the current time formatted as time only. It should display something similar to 12439d 16:58:58.

```
display formatTimeOnly('now')
```

formatTimeOnly

Formats this Time object as a time only, with no time zone correction, applying the specified style.

Arguments

- Time - The Time object to be formatted.
- Int - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the representation of the Time object formatted as time only, applying the specified style.

Example

The following example only displays the time representation of the current time in the four available styles.

```
defaultTime as String = "Default format --> " +
    formatTimeOnly('now', intervalStyle : Time.DEFAULT)

fullTime as String = "Full format --> " +
    formatTimeOnly('now', intervalStyle : Time.FULL)

longTime as String = "Long format --> " +
    formatTimeOnly('now', intervalStyle : Time.LONG)

shortTime as String = "Short format --> " +
    formatTimeOnly('now', intervalStyle : Time.SHORT)

display defaultTime + "\n\n" +
    fullTime + "\n\n" +
    longTime + "\n\n" +
    shortTime + "\n\n"
```

formatTime

Formats the time component of a Time object with a specified style.

Arguments

- Time - The Time object to be formatted.
- Int timeStyle - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A String containing the time component of the Time object formatted with the specified style.

Example

The following example displays the time component of the current time formatted with the four available styles.

```
defaultTime as String = "Default format --> " +
    formatTime('now', timeStyle : Time.DEFAULT)

fullTime as String = "Full format --> " +
    formatTime('now', timeStyle : Time.FULL)

longTime as String = "Long format --> " +
    formatTime('now', timeStyle : Time.LONG)

shortDate as String = "Short format --> " +
    formatTime('now', timeStyle : Time.SHORT)

display defaultTime + "\n\n" +
    fullTime + "\n\n" +
    longTime + "\n\n" +
    shortTime + "\n\n"
```

format

Formats a Time object with the default mask.

Arguments

- Time - the Time object to be formatted.

Returns

A string containing the representation of the Time object formatted with the default mask.

Example

The following example displays the current time formatted with the default mask.

```
display format('now')
```

format

Formats a Time object with a specified date style and time style.

Arguments

- Time - The Time object to be formatted.
- Int dateStyle - The style to be applied to the date component of the Time object. Available styles are: Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.
- Int timeStyle - The style to be applied to the time component of the Time object. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

A string containing the representation of the Time object whose date was formatted with the specified date style, and whose time was formatted with the specified time style.

Example

The following example displays the current time with its date in full format style and its time in short format style first, then the same time with its date in short format style and its time in full format style.

```
fullDateShortTime as String = "Full date, short time: " +  
    format('now', dateStyle : Time.FULL,  
          timeStyle : Time.SHORT)  
  
shortDateFullTime as String = "Short date, full time: " +  
    format('now', dateStyle : Time.SHORT,  
          timeStyle : Time.FULL)  
  
display fullDateShortTime + "\n\n" + shortDateFullTime
```

format

Formats a Time object by applying a specified formatter.

Arguments

- Time - The Time object to be formatted.
- Java.Text.DateFormat formatter - The formatter to be applied in order to format the Time object.

Returns

A String containing the representation of the Time object formatted

by applying the specified formatter.

Example

The following example displays the current Time formatted with the formatter passed by arguments.

```
display format('now', formatter : DateFormat.getInstance())
```

format

Formats a Time object with a specified formatter using the provided time zone and locale.

Arguments

- Time - The Time object to be formatted.
- Java.Text.DateFormat formatter - The formatter to be applied in order to format the Time object.
- Java.Util.TimeZone timeZone - The time zone to apply to the Time object when formatting it.
- Java.Util.Locale locale - The locale to apply to the Time object when formatting it.

Returns

A String containing the representation of the Time object formatted by applying the specified formatter and the time zone and locale provided.

Example

The following example displays the current Time formatted applying

the formatter passed by arguments, GMT-10 time zone and French locale.

```
display format('now', formatter : DateFormat.getInstance(),
    timeZone : TimeZone.getTimeZone(arg1 : "GMT-10"),
    locale : Java.Util.Locale.FRANCE)
```

format

Formats a Time object with a specified mask.

Arguments

- Time - The Time object to be formatted.
- String mask - A string containing the mask to apply in order to format the Time object.

Returns

A String containing the representation of the Time object formatted with the specified mask. The String should be written according to the patterns and rules described below.

The following pattern letters are defined (all other characters from 'A' to 'Z' and from 'a' to 'z' are reserved):

Letter	Date or Time Component	Presentation	Example
G	Era designator	Text	AD
y	Year	Year	1996; 96
M	Month in year	Month	July; Jul; 07
w	Week in year	Number	27

Letter	Date or Time Component	Presentation	Example
W	Week in month	Number	2
D	Day in year	Number	189
d	Day in month	Number	10
F	Day of week in month	Number	2
E	Day in week	Text	Tuesday; Tue
a	Am/pm marker	Text	PM
H	Hour in day (0-23)	Number	0
k	Hour in day (1-24)	Number	24
K	Hour in am/pm (0-11)	Number	0
h	Hour in am/pm (1-12)	Number	12
m	Minute in hour	Number	30
s	Second in minute	Number	55
S	Millisecond	Number	978
z	Time zone	General time zone	Pacific Standard Time; PST; GMT-08:00
Z	Time zone	RFC 822 time zone	-0800

Pattern letters are usually repeated, as their number determines the exact presentation:

- Text: For formatting, if the number of pattern letters is 4 or more, the full form is used. Otherwise, a short or abbreviated form is used if available. For parsing, both forms are accepted,

independent of the number of pattern letters.

- **Number:** For formatting, the number of pattern letters is the minimum number of digits, and shorter numbers are zero-padded to this amount. For parsing, the number of pattern letters is ignored unless it's needed to separate two adjacent fields.
- **Year:** For formatting, if the number of pattern letters is 2, the year is truncated to 2 digits. Otherwise, it is interpreted as a number.

Example

The following example displays the current time formatted with the mask defined by the String passed by arguments. It should display something like: Thu 22 01 2004 04:31:48:975 PM ART

```
display 'now'.format("E dd MM yyyy hh:mm:ss:SS a z")
```

getDateFormat

Returns an appropriate DateFormat based on the parts needed and a style.

Arguments

- **Int parts** - The needed parts. These could be Time.DATE_ONLY, Time.TIME_ONLY, or Time.DATE_TIME.
- **Int style** - The formatting style. Available styles are Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

Returns

The format based on the required parts and style.

Example

The following example uses the function `getDateFormat` with different needed parts and style to format the current time.

```
fullDateOnly as String
fullDateOnly = "Date only - Full format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_ONLY,
    style : Time.FULL))

shortTimeOnly as String
shortTimeOnly = "Time only - Short format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.TIME_ONLY,
    style : Time.SHORT))

longDateTime as String
longDateTime = "Date time - Long format: " +
  'now'.format(Time.getDateFormat(
    parts : Time.DATE_TIME,
    style : Time.FULL))

display fullDateOnly + "\n\n" + shortTimeOnly + "\n\n" +
  longDateTime
```

getDateFormat

Returns an appropriate `DateFormat` based on the required parts, style, and locale.

Arguments

- `Int parts` - The needed parts. These could be `Time.DATE_ONLY`, `Time.TIME_ONLY`, or `Time.DATE_TIME`.
- `Int style` - The formatting style. Available styles are

Time.DEFAULT, Time.FULL, Time.LONG, and Time.SHORT.

- Java.Util.Locale - The locale the date formatter will have to apply.

Returns

Returns an appropriate DateFormat based on the required parts, style, and locale.

Example

The following example uses the function getDateFormat with different needed parts and style, and french locale, to format the current time.

```
fullDateOnly as String
fullDateOnly = "Date only - Full format: " +
    'now'.format(Time.getDateFormat(
        parts : Time.DATE_ONLY,
        style : Time.FULL, Java.Util.Locale.FRANCE))

shortTimeOnly as String
shortTimeOnly = "Time only - Short format: " +
    'now'.format(Time.getDateFormat(
        parts : Time.TIME_ONLY,
        style : Time.SHORT, Java.Util.Locale.FRANCE))

longDateTime as String
longDateTime = "Date time - Long format: " +
    'now'.format(Time.getDateFormat(
        parts : Time.DATE_TIME,
        style : Time.FULL, Java.Util.Locale.FRANCE))

display fullDateOnly + "\n\n" + shortTimeOnly + "\n\n" +
    longDateTime
```

getEaster

Calculates Easter day for a specified year.

Arguments

- Int y - The year from which you would like to know Easter's date.

Returns

A Time object containing Easter date.

Example

The following example displays Easter date for the year 2004.

```
display Time.getEaster(y : 2004)
```

getMonthName

Returns the name of the specified month.

Arguments

- Int monthNum - The number which identifies the month from which you would like to know the name. This number may vary between 1 (January) and 12 (December).

Returns

The name of the specified month.

Example

The following example displays the name of month 2 (February).

```
display Time.getMonthName(monthNum : 2)
```

getWeekdayName

Returns the name of the specified weekday.

Arguments

- Int dayNum - The number which identifies the day from which you would like to know the name. This number may vary between 0 (Sunday) and 6 (Saturday).

Returns

The name of the specified day.

Example

The following example displays the day 3 (Wednesday).

```
display Time.getWeekdayName(dayNum : 0)
```

hour

This function work exactly the same way as hourOfDay Please see hourOfDay for further details.

max

Returns the greater time between two Time objects.

Arguments

- Time - A Time object.
- Time b - Another Time object.

Returns

The greater Time object.

Example

The following example displays the greater Time between the current time and the date the man landed on the moon. The current time is returned.

```
manOnMoon as Time
manOnMoon = '1969-07-21 02:56:00 -00 '
display max(manOnMoon, b : 'now')
```

min

Returns the smaller Time object between two Time objects.

Arguments

- Time - A Time object.
- Time b - Another Time object.

Returns

The smaller Time object.

Example

```
manOnMoon as Time
manOnMoon = '1969-07-21 02:56:00 -00 '
display min(manOnMoon, b : 'now')
```

monthsSince

Returns the number of months passed since a given Time object.

Arguments

- Time - The given Time object.

Returns

An Int value representing the number of months passed since the given Time object.

Example

```
xMas2003 as Time  
xMas2003 = '2003-12-25'  
display monthsSince('now', t : xMas2003)
```

roundToSeconds

Rounds a Time object to a seconds' precision.

Arguments

- Time - The given Time object.

Returns

The given Time object with one seconds' precision.

Example

The following example creates a Time object with seconds and

milliseconds, then it rounds this object to one seconds' precision and displays the result.

```
time as Time
time = '2004-01-23 17:21:30.2356 -00'
display roundToSeconds(time)
```

Interval Functions

abs

Returns the absolute value of a given interval.

Arguments

- Interval - The given Interval object.

Returns

The absolute value of a given Interval object.

Example

The following example calculates an Interval by subtracting the current time from the first day of year 2000. Then, it displays the absolute value of the resulting interval.

```
year2k as Time
year2k = '2000-01-01 00:00:00'
interval = year2k - 'now'
display interval
display abs(interval)
```

addDays

Adds a specified amount of days to an Interval object.

Arguments

- Interval - The interval to which days will be added.
- Int i - The amount of days to be added to the Interval object.

Returns

An Interval object resulting from adding the specified amount of days to the given Interval object.

Example

The following example creates an Interval variable named holidays and another named newHolidays, which is the result of adding 10 days to holidays. Then, it displays both the original variable and the one resulting from adding 10 days to the original variable.

```
holidays as Interval
holidays = '15d20h00m'

updatedHolidays as Interval
updatedHolidays = addDays(holidays, i : 10)

display "original holidays: " + holidays +
  "\n\n updated holidays: " +
  updatedHolidays
```

addHours

Adds a specified amount of hours to an Interval object.

Arguments

- Interval - The interval to which hours will be added.
- Int i - The amount of hours to be added to the Interval object.

Returns

An Interval object resulting from adding the specified amount of hours to the given Interval object.

Example

The following example creates an Interval variable named `deliveryTime` and another named `newDeliveryTime`, which is the result of adding 12 hours to `deliveryTime`. Then, it displays both the original variable and the one resulting from adding 12 hours to the original variable.

```
deliveryTime as Interval
newDeliveryTime as Interval

deliveryTime = '1d00h00m'
newDeliveryTime = addHours(deliveryTime, i : 12)

display "original deliveryTime: " + deliveryTime +
        "\n\n new deliveryTime: " + newDeliveryTime
```

addMicroSeconds

Adds a specified amount of microseconds to an Interval object.

Arguments

- Interval - The interval to which microseconds will be added.
- Int i - The amount of microseconds to be added to the Interval object.

Returns

An Interval object resulting from adding the specified amount of microseconds to the given Interval object.

Example

The following example creates an Interval variable named `retry` and another named `largerRetry`, which is the result of adding 222 hours to `retry`. Then, it displays both the original variable and the one resulting from adding 222 microseconds to the original variable.

```
retry as Interval
retry = '1m30.600000s'

largerRetry as Interval
largerRetry = addMicroSeconds(retry, i : 222)

display "old retry: " + retry + "\n\nnew retry: " +
    largerRetry
```

addMinutes

Adds a specified amount of minutes to an Interval object.

Arguments

- Interval - The interval to which minutes will be added.
- Int i - The amount of minutes to be added to the Interval object.

Returns

An Interval object resulting from adding the specified amount of minutes to the given Interval object.

Example

The following example creates an Interval variable named `breakTime` and another named `newBreakTime`, which is the result of adding 25 minutes to `breakTime`. Then, it displays both the original variable and the one resulting from adding 25 minutes to the original variable.

```
breakTime as Interval
breakTime = '1h20m00s'

newBreakTime as Interval
newBreakTime = addMinutes(breakTime, i : 25)

display "old break-time: " +breakTime +
        "\n\n new break-time: "+ newBreakTime
```

addMonths

Adds a specified amount of months to an Interval object.

Arguments

- Interval - The interval to which months will be added.
- Int i - The amount of months to be added to the Interval object.

Returns

An Interval object resulting from adding the specified amount of months to the given Interval object.

Example

The following example defines an Interval `fishingSeason`. Then, it creates a new Interval named `newFishingSeason`, which is the result of adding a month to `fishingSeason`. Both the original interval and the result of the addition are displayed.

```
fishingSeason as Interval
fishingSeason = '1M20d'

newFishingSeason as Interval
newFishingSeason = addMonths(fishingSeason, i : 1)

display "original fishingSeason: " + fishingSeason +
"\n\nnew fishingSeason: " + newFishingSeason
```

addYears

Adds a specified number of years to an Interval object.

Arguments

- Interval - The Interval object to which years will be added.
- Int i - The amount of years to be added to the Interval object.

Returns

The Interval object resulting from adding the specified amount of years to the given Interval.

Example

```
licensePeriod as Interval
licensePeriod = '1Y6M'

newLicensePeriod as Interval
newLicensePeriod = addYears(licensePeriod, i : 1)

display "original license period: "+ licensePeriod +
"\n\nnew license period: " + newLicensePeriod
```

format

Returns a String representation of the given interval. This String representation is based on the current locale.

Arguments

- Interval - The given Interval object.

Returns

A String representation of the given Interval based on the current locale.

Example

```
interval as Interval  
interval = '2Y6M15d12h30m30s'  
display format(interval)
```

intValue

Returns the Int value of an Interval objects. This value represents the number of seconds in the interval.

Arguments

- Interval - The Interval object.

Returns

An Int representing the number of seconds in the given interval.

Example

The following example displays the Int representation, that is to say, the number of seconds of an interval of 1 hour 30 minutes and 20 seconds.

```
display intValue('1h30m20s')
```

max

Returns the greater of two Interval objects.

Arguments

- Interval - The given Interval object.
- Interval b - Another Interval object.

Returns

The greater of the two Interval objects.

Example

The following example displays the greater Interval between 1 hour and 20 minutes and 1 hour and 35 minutes.

```
display max('1h20m', b : '1h35m')
```

min

Returns the smaller of two Interval objects.

Arguments

- Interval - The given Interval object.
- Interval b - Another Interval object.

Returns

The smaller of the two Interval objects.

Example

The following example displays the smaller Interval between 1 hour and 20 minutes and 1 hour and 35 minutes.

```
display min('1h20m', b : '1h35m')
```

sleep

Causes the current BP-method to sleep for the specified amount of time. Note that you cannot sleep past the current timeout. While the method is *sleeping* the timeout period is still running, therefore the *sleep* is applicable for the defined interval or until timeout, whatever happens first.

Arguments

Interval - Time to wait

Example

The following example pauses the execution for 5 seconds

```
sleep('5s')
```

Chapter 10. Arrays

Arrays

An array is a collection of values of the same type. Each element of the array is identified with an index or a key. Any type that can be used to declare a variable can be used to declare an array, even another array.

Types of arrays

FuegoBPM Studio supports the following types of arrays:

- Indexed Arrays
- Associative Arrays

Indexed arrays are indexed by consecutive positive integers, starting from 0 (zero).

Associative arrays may be indexed by any type, although certain types are better suited to be used as indexes.

For further information about Arrays, please refer to sections:

- Manipulating arrays
- Array Functions
- Array Attributes
- Array Procedures
- Mapping members

Indexed Arrays

Indexed arrays are collections of elements of the same type, indexed by an Int starting in zero.

Declaration


Indexed arrays are declared similarly to other types of variables but with the difference that you put a set of '[]' brackets at the end of the type in order to indicate it is an array.

Example:

```
names as String[]
```

The code shown above declares an associative array named *names* with elements of String type.

Note

 Avoid using ArrayList, HashMap, Iterator as FuegoBPM provides a built-in support for arrays

Initializing an array

You can use in-line arrays for initialization:

```
names as String[]  
names = ["John", "Peter", "Mary"]
```

The code shown above initializes the array *names* with the String "John" at position 0 (zero), "Peter" at position 1 (one) and "Mary" at position 2 (two).

You can also use the append operator ([]):

```
names as String[]  
names[] = "John"  
names[] = "Peter"  
names[] = "Mary"
```

This code also initializes the array *names* with the String "John" at position 0 (zero), "Peter" at position 1 (one) and "Mary" at position 2 (two).

The append operator ([]) adds an element to the end of the array.

You can even use the indexes to add elements:

```
names as String[]  
names[0] = "John"  
names[1] = "Peter"  
names[2] = "Mary"
```

Let's examine this example, line by line:

On the first line, you declare the *names* array, which is initially empty, or what is the same, its length is 0 (zero):

```
names as String[]
```

On the second line, you refer to the element 0 (zero), which does not exist, but it is equal to the length of the array. Since you are accessing it for assignment, it can be safely assumed that you want to enlarge the array, and it does so.

```
names[0] = "John"
```


Now the array has 1 element: ["John"]

The following steps are analogous to the previous:

```
names[1] = "Peter"  
names[2] = "Mary"
```

In the end, the array has 3 elements: ["John", "Peter", "Mary"]

Note that the following will not work because the index you request is out of bounds:

```
names as String[]  
names[2] = "Mary"
```

Accessing elements

In order to access an element of an array, you use the variable name suffixed with the index you want:

```
names as String[]  
names = ["John", "Peter", "Mary"]  
display names[1]
```

Note that the index can be an expression:

```
names as String[]  
names = ["John", "Peter", "Mary"]  
display names[names.length - 1]
```

```
names as String[]  
for each element in names do  
...  
end
```

Associative Arrays

Associative arrays are arrays that map one set of values to another. The types of their indexes can be different from `Int`. The index values are not required to be contiguous.

Declaration

Associative arrays are similarly declared to indexed arrays, but with the difference that you specify the type of the index.

Example:

```
ages as Int[String]
```

The code above declares an associative array named `ages`, which is of type `Int` and is indexed by strings.

Initializing an array

You can use in-line arrays for initialization. They are similar to indexed arrays, but you must specify the index since the index is arbitrary:

```
ages as Int[String]  
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
```

The code above initializes the array `'ages'`, associating the value 23

to the key "John", the value 42 to the key "Peter", and the value 29 to the key "Mary".

You can also use the keys to add elements:

```
ages as Int[String]
ages["John"] = 23
ages["Peter"] = 42
ages["Mary"] = 29
```

The code above also initializes the array 'ages', associating the value 23 to the key "John", the value 42 to the key "Peter", and the value 29 to the key "Mary".

Accessing elements

Similar to indexed arrays, the elements of an indexed array can be accessed by key, passing the key between square brackets:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
display ages["John"]
```

If you pass a nonexistent key, the element returned is **null**.

Ordered arrays

The keys of an associative array can be automatically ordered. To do so, the array must be declared using 'ordered' before the index declaration.

Example:

```
ages as Int[ordered String]
```

The type used for the index must be comparable, that is to say, a default ordering is defined for such type.

The following example shows each of the keys, using an unordered array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
for key in ages do
  display key
end
```

Note that the order of the elements is not guaranteed; they will be iterated in any order.

The following example is the same as above, but using an ordered array:

```
ages as Int[ordered String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]
for key in ages do
  display key
end
```

The keys are listed in ascending order according to the keys default ordering.

Manipulating arrays

Change Elements

To change an array element, you can just access it with its list number and assign it a new value:

```
products = ["A", "B", "C"]  
products[2] = "D"
```

This changes the value of the element with the list number two (remember, arrays start counting at zero, so the element with the list number two is actually the third element.) As we changed "C" to "D", the array now contains "A", "B" and "D".

Add Elements

Now, suppose that we want to add a new element to the array. We can add a new element in the last position by just assigning a value to the next position. If it does not exist, it is added at the end:

```
products = ["A", "B", "C"]  
products[3] = "D"
```

Now, the array has four elements: "A", "B", "C" and "D".

It is not necessary to know the array length to add an element at the end. In order to do it, the 'add' ([]) operator or the 'extend' method can be used.

Example, the following method is equivalent to the previous one:

```
products = ["A", "B", "C"]  
extend products          // add at the end  
    using "D"
```

Delete Elements

Elements can be deleted from an array by using the operator

'delete':

```
products = ["A", "B", "C"]  
delete products[0] // delete first element
```

Now, the array has two elements "B", "C".

Find Elements

The 'in' operator can be used to check if an element is contained in an array. The following code checks whether "A" is contained in the array 'products'.

```
products = ["A", "B", "C"]  
if "A" in products then  
    display "'products' contains the element 'A'"  
end
```

Now, if you want to get the index of the first occurrence of an element:

```
products = ["A", "B", "A", "C"]  
index = indexOf(products, "A")  
  
if index != -1 then  
    display "'A' is located at position : " + index  
end
```

Last examples will show the index 0. Instead, if you want to find the last occurrence, 'lastIndexOf' can be used:

```
products = ["A", "B", "A", "C"]
```

```
index = lastIndexOf(products, "A")  
if index != -1 then  
    display "'A' is located at position : " + index  
end
```

Array Functions

avg

Calculates the average value of the data contained in an array. Its behavior is defined for numeric element types only.

This function is parametric, which means that its return type depends on the element type of the array passed as an argument.

Arguments

- Array: the array on which the function operates.

Returns

- The average of all the element types.

Example

```
array as Decimal [] = [10.49, 13.78, 33.99]  
display avg(array)
```

The expected result is 19.42.

count

Counts the number of non-null elements contained in an array.

Arguments

- Array: the array on which the function operates.

Returns

- Int: The number of non-null elements contained in the array.

Example

```
array as Int[]=[22,33,null,55]  
display count(array)
```

The expected result is 3.

indexOf

This function returns the index of the first occurrence of an element in the array.

Arguments

- Array: the array on which the function operates.
- Any: the element you are looking for.

Returns

- Int: the index of the element or -1 if not found.

Example

```
array as String[] = ["Hello","world","!!!"]  
display indexOf(array, "world")
```

The expected result is 1.

```
array as String[] = ["Hello","world","!!!"]  
display indexOf(array, "happy")
```

The expected result is -1.

lastIndexOf

This function returns the index of the element's last occurrence.

Arguments

- Array: the array on which the function operates.
- Any: the element you are looking for.

Returns

- Int: the index of the element or -1 if not found.

Example

```
array as String[] = ["Hello","!!!","world","!!!"]  
display lastIndexOf (array, "!!!")
```

The expected result is 3.

```
array as String[] = ["Hello","!!!","world","!!!"]  
display lastIndexOf (array, "happy")
```

The expected result is -1.

length

This function returns the length of the array.

Arguments

- Array: the array on which the function operates.

Returns

- Int: the array's length.

Example

```
array as Int[] = [22,33,44,55]  
display length(array)
```

The expected result is 4.

max

This function returns the maximum element of the array. The element type must have a defined default ordering.

This function is parametric, which means that its return type depends on the element type of the array passed as an argument.

Arguments

- Array: the array on which the function operates.

Returns

- the maximum element of the array.

Example

```
array as Int[]=[22,33,44,55]  
display max(array)
```

The expected result is 55.

min

This function returns the minimum element of the array. The element type must have a default ordering defined.

This function is parametric, which means that its return type depends on the element type of the array passed as an argument.

Arguments

- Array: the array on which the function operates.

Returns

- the minimum element of the array.

Example

```
array as Int[]=[22,33,44,55]  
display min(array)
```

The expected result is 22.

sum

Calculates the sum of all the array's elements. Its behavior is defined for numeric element types only.

This function is parametric, which means that its return type depends on the element type of the array passed as an argument.

Arguments

- Array: the array on which the function operates.

Returns

- The sum of all the element types.

Example

```
array as Int[]=[112,3233,454,595]  
display sum(array)
```

The expected result is 4394.

Array Attributes

first

This attribute returns the first element of an array. This attribute is parametric, which means that its type depends on the element type of the array on which it is applied.

Returns

- the first element of the array or null if the array is empty.

Example

```
array as String[]= ["Hello","World","!!!"]  
display array.first
```

The expected result is "Hello".

last

This attribute returns the last element of an array. This attribute is parametric, which means that its type depends on the element type of the array on which it is applied.

Returns

- the last element of the array or null if the array is empty.

Example

```
array as String[] = ["Hello", "World", "!!!"]  
display array.last
```

The expected result is "!!!".

Array Procedures

sort

Sorts the elements of the array. The element type must have a defined default ordering.

Arguments

- Array: the array on which the procedure operates.

Example

```
array as Int [] = [7, 3, 5, 2]  
sort array  
display array
```

The expected result is [2, 3, 5, 7].

clear

Clears all the elements of the array.

Arguments

- Array: the array on which the procedure operates.

Example

```
array as Int [] = [7, 3, 5, 2]
clear array
display array
```

The expected result is [].

insert

Inserts an element in the specified position.

Arguments

- Indexed Array: An indexed array on which the function operates.

Example

```
array as Int[] = [2, 7, 3, 4]
insert array
    using int = 1,
        value = 9
display array
```

The expected result is [2, 9, 7, 3, 4].

Mapping members

All of the attributes of an array's element type are mapped to the array. This means that if you have an array of *MyComponent* and *MyComponent* has an attribute called *name* of String type, you can do the following:

```
array as MyComponent[]
names as String[]
// ... initialize array ...
names = array[].name
```

This has the effect of having an attribute called *name* on the array of String type.

The equivalent code without using member mapping is as follows:

```
array as MyComponent[]
names as String[]
// ... initialize array ...
for each component in array do
    names[] = component.name
end
```

The following is an example using Interval:

```
intervals as Interval[]
intervals = ['1d2h', '2d3h', '5d6h']
display intervals[].hours
```

This displays an Int array containing [2, 3, 6].

Chapter 11. Enumerations

Enumerations

Enumerations are sets of related integer constants, where each value has a name.

FuegoBPM Studio has built-in support for enumerations (sequential and non-sequential). These are some enumerations' properties:

- Type safe: The compiler checks that the values belong to the enumeration.
- User-Friendly: Understanding a piece of code that uses them is easier. Enumerations have a name that indicates what they represent.
- Improved Performance: Comparison of enumeration values is reduced to a comparison between integers.

Using Enumerations

Enumerations can be used with a qualified name:

```
action = Action.SKIP
```

or without qualification, when the type of the enumeration can be inferred from the context:

```
action = SKIP
```

To check the value of an Enumeration, you can use the **is** operator:

```
if action is CANCEL then
  //Do something
end
```

or if you prefer, the multi-path conditional statement:

```
case action
when SKIP then
  // Process SKIP
when CANCEL then
  // Process CANCEL
else
  // Handle all other values
end
```

Creating Enumerations

For further information on creating enumerations, refer to: Enumerations.

Chapter 12. Statements

Statements

Sequential execution is the most basic path that a method execution can take. The method starts its execution on the first line of the code, followed by the next and continues until the final statement in the code has been executed. This approach works fine for very simple tasks but tends to lack usefulness since it can only handle one situation. Like humans, most programs need to be able to decide what to do in response to changing circumstances. By controlling a code's execution path, a specific piece of code can then be used to handle more than one situation intelligently.

Control Statements are used to control the sequence of statement execution.

There are other statements, such as Assignment Statement, which allow you to set values to variables, and Interactive Statements, which allow you to interact with the current participant.

Interactive Statements

Input statement

Input statements allow you to invite the current participant to enter information that is required by the Method logic.

The following types are available:

- Basic input, which builds a simple form for entering data.
- Interceptor input, which intercepts one or more web pages fetching data from the different form fields.
- Fuego Object input, which displays a presentation of a Fuego Object.

Basic input and **Fuego Object input** can be implemented with screenflows. This is strongly recommended. See Screenflow's documentation for detailed information.

Warning



If you use the **input** statement in a BP-Method in a task, when the design is checked, a warning is thrown. It is recommended to use a screenflow or the *relay to* option, as the input statement is not applicable if you run the process in an EJB based Engine.

Syntax

Basic input

```
input "field label" basicReference
    [( {option} [= {value}], ... )]
    [ in [{valid values}] ] [, ...]
    [using
        title = "{title}",
        buttons = [{button labels}],
        cancelButton = "{button label}"
    ]
    [returning
        {selected button} = selection
    ]
```

Interceptor input

```
input "field label" basicReference
    [ in [{valid values}] ] [, ...]
    [using
        title = "{title}",
        buttons = [{button labels}],
        htmlForm = "{intitial URL}",
        until = "{stop condition}",
```

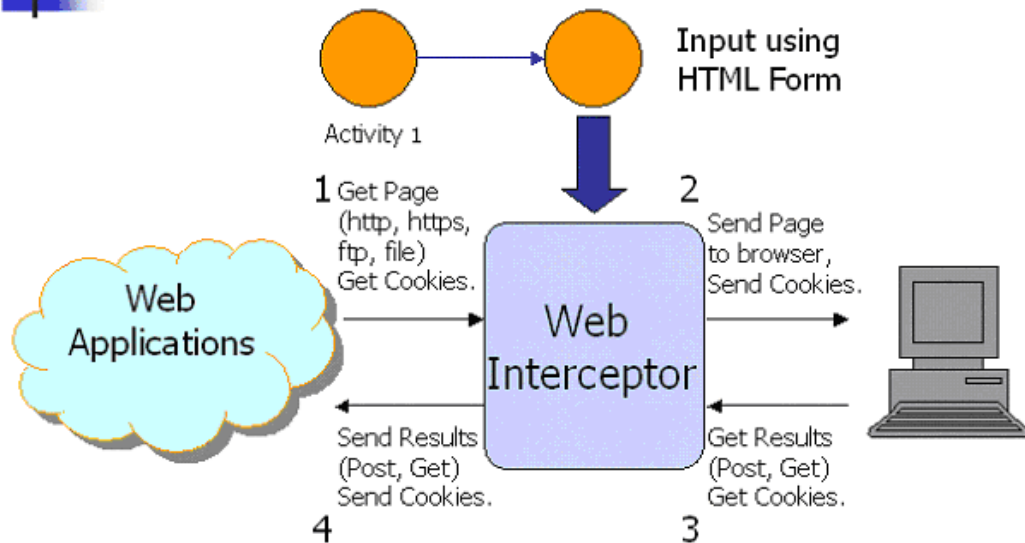
```
links = "{ intercept | popup | clear }",
userControl = { true | false },
cookies = [ { cookies } ]
]
[returning
  {selected button} = selection,
  {cookie map} = cookies
]
```

The main functionality of the Web Interceptor can be simplified in these 3 topics:

- Interact with existing Web Applications.
- Allow BP-Method access forms in HTML, JSP, ASP.
- Support any form control as Lists, Text Areas, Radio Buttons, Fields.

How does the Web Interceptor work?

Information Flow



Web Interceptor basically works over HTTP(S), FTP and FILE resources.

The referenced URL (*htmlForm*) can be:

- http://...
- https://...
- ftp://...
- file://...

Fuego Object input

```
input basicReference
  [using
    selectedPresentation = "{presentation name}"
```

```
    ]  
    [returning  
      {selected button} = selection  
    ]
```

Arguments

Attribute: title

Type: String

Description: The input form's title.

Attribute: buttons

Type: String[]

Description: An array containing the labels of the buttons you want to be displayed on the form.

Attribute: cancelButton

Type: String

Description: button that acts ignoring all changes and avoiding the input of required fields.

Attribute: htmlForm

Type: String

Description: Full or relative URL to the initial page to be intercepted.

For relative files as **.html**, **.jsp** or **.asp** to be intercepted have to be copied into the Fuego's installation directory, in studio or enterprise/webapps/portal/*dirname*.

The **htmlForm** attribute is composed by:

http://host:port/projectname/dirname/file.[html/jsp/asp]",
where:

host:port is the host and port in which you run the Work Portal.

projectname is the name of your project,

dirname is the name of the directory you created in **studio or enterprise/webapps/portal/**

For example:

htmlForm =
"http://localhost:9595/InterceptorCase01/TestInterceptor/classRegister.html",

InterceptorCase01 is the name of the project,

TestInterceptor is the directory in studio/webapps/portal, and

classRegsiter.html is the html page to be intercepted.

Attribute: until

Type: String

Description: Stop condition that indicates the interceptor when to stop intercepting pages. It is of the form: *pattern@location*

Where *location* is a full or partial url that marks the end of interceptions and *pattern* is a string that is sought for when *location* is reached. (pattern@URL or pattern@file name)

If not found, interception continues.

Some examples for stop conditions are:

- post.asp?id=query
- http://mysite/Poster/post.asp
- Congratulations@post.asp

- Congratulations@http://mysite/Poster/post.asp

Attribute: links

Type: String




Description: This parameter determines what happens when the intercepted page contains links and the user clicks on one. It takes one of the following values:

- popup: the content of the link will be displayed in a new window.
- clear: the link will be removed.
- intercept: the link will be intercepted and displayed in the same window.

It will continue to intercept until the specified criteria has been met or the user hits "Stop" in the navigational control (see: userControl attribute.)

Attribute: userControl

Type: Bool




Description: If true, the navigational control (  ) will be displayed in the intercepted page.

One of the purposes of using Web interceptor is to set and get information from the pages you are intercepting or navigating. This basically implies that you are able to go through a sequence of HTML, JSP or ASP pages intercepting and collecting information that will later be used in the BP-Method logic. The navigational tool allows the end user go back and forward through the pages until the stop criteria is reached.

In order to show the navigational tool in the intercepted pages you

need to set “userControl” to true. If you set true to *links*, then *userControl* takes the value true automatically and the navigational toolbar is displayed.

The buttons showed in the navigational panel are:

- Go Back : Goes to the previous intercepted page.
- Finish Interceptor : Stops the interception and returns the control to the BP-Method.
- First page : It starts again the interception from the first page.

Attribute: cookies

Type: java.Object[java.Object]

Description: Associative array that contains a collection of cookies used during interception. The keys and values are of String type.

Attribute: selection

Type: String

Description: Returns the label of the button that caused the form to be dismissed.

Attribute: selectedPresentation

Type: String

Description: The name of the desired presentation for the Object, if this attribute is not specified, the default presentation will be used

Field options

The following table contains a list of the **options** that can be passed to an input field:

Option	Required Type	Description
date	Time	displays a Time as date-only.
time	Time	displays a Time as time-only.
readonly	Any	the field is displayed, but cannot be modified.
password	String	the field is displayed as a password field.
required	Any	The field cannot be null.
textarea	String	displays an area to enter a text.

Remarks

For the Fuego Object input, the `selectedPresentation` attribute is only valid if the `basicReference` is a Fuego Object. If it is not, a field label will be synthesized and the input will behave as a basic input.

If the `selectedPresentation` attribute is missing, the default presentation of the Fuego Object will be displayed.

When you specify a partial URL in any of the fields that take one, the URL is relative to the portal in which the input statement is displayed.

For an Interceptor input, the field name must match the name of a field in the form being intercepted. If it does not match, the variable will be left empty.

Input Examples

Basic Input

```
creditCardNo = ""
acceptedTypes = ["visa" : "Visa", "master" : "Mastercard",
    "amex" : "American Express"]
creditCardType = "visa"
firstName = ""
lastName = ""
expiration = 'now'
comments = ""
input "First Name:" firstName (required),
    "Last Name:" lastName (required),
    "Credit card type:" creditCardType (required)
    in acceptedTypes,
    "Credit card No.:" creditCardNo,
    "Expiration Date:" expiration,
    "Additional Comments:" comments (textarea)
using
    title = "Enter payment info",
    buttons = ["Ok", "Cancel"]
```

Interceptor Input

```
googleQuery = ""
input "q" googleQuery
using
    htmlForm = "http://www.google.com",
    links = "clear"
```

Refer to Web Interceptor for more examples.

Fuego Object Input

```
//Order is a Fuego Object with a presentation called
//'auditView'
input order
using
    selectedPresentation="auditView"
```

Display Statement

The display statement, as its name implies, allows you to display information to the user and to get feedback based on the choice of buttons the user selected.

The **Display** can be implemented with screenflows. This is strongly recommended. See Screenflow's documentation for detailed information.

Warning



If you use the **display** statement in a BP-Method in a task, when the design is checked, a warning is thrown. It is recommended to use a screenflow, as the display statement is not applicable if you run the process in an EJB based Engine.

Syntax

Basic Display

```
display {object}
[ using
  [title = "{title}", ]
  [type = "{error | question | warning | info | plain}", ]
  [options = {options}, ]
  [default = {default button}]
]
[ returning {selected button} = selection ]
```

Fuego Object Display

```
display {fuego object}
[ using selectedPresentation = "{presentation name" ]
```

This form of display shows a Fuego Object presentation as read-only.

Arguments

The following is the list of the arguments that can be passed to a display statement:

Argument: title

Type: String

Description: Title of the display window/page.

Argument: type

Type: String

Description: Kind of display. The icon will be chosen based on this argument. The default value is "plain".

Argument: options

Type: String[]

Description: Array of strings containing the labels of the buttons you want to display.

Argument: default

Type: String

Description: Which of the buttons is returned in case the display is canceled.

Argument: selectedPresentation

Type: String

Description: Name of the presentation used to display a Fuego Object. If left unspecified, the default presentation will be used.

Display Examples

Basic Display

```
selectedButton as String
display "Should we try again?"
  using title = "Confirm",
        type = "question",
        buttons = ["Yes", "No"],
        default = "No"
  returning selectedButton = selection

if selectedButton = "Yes" then
  //Retry
end
```

Fuego Object Display

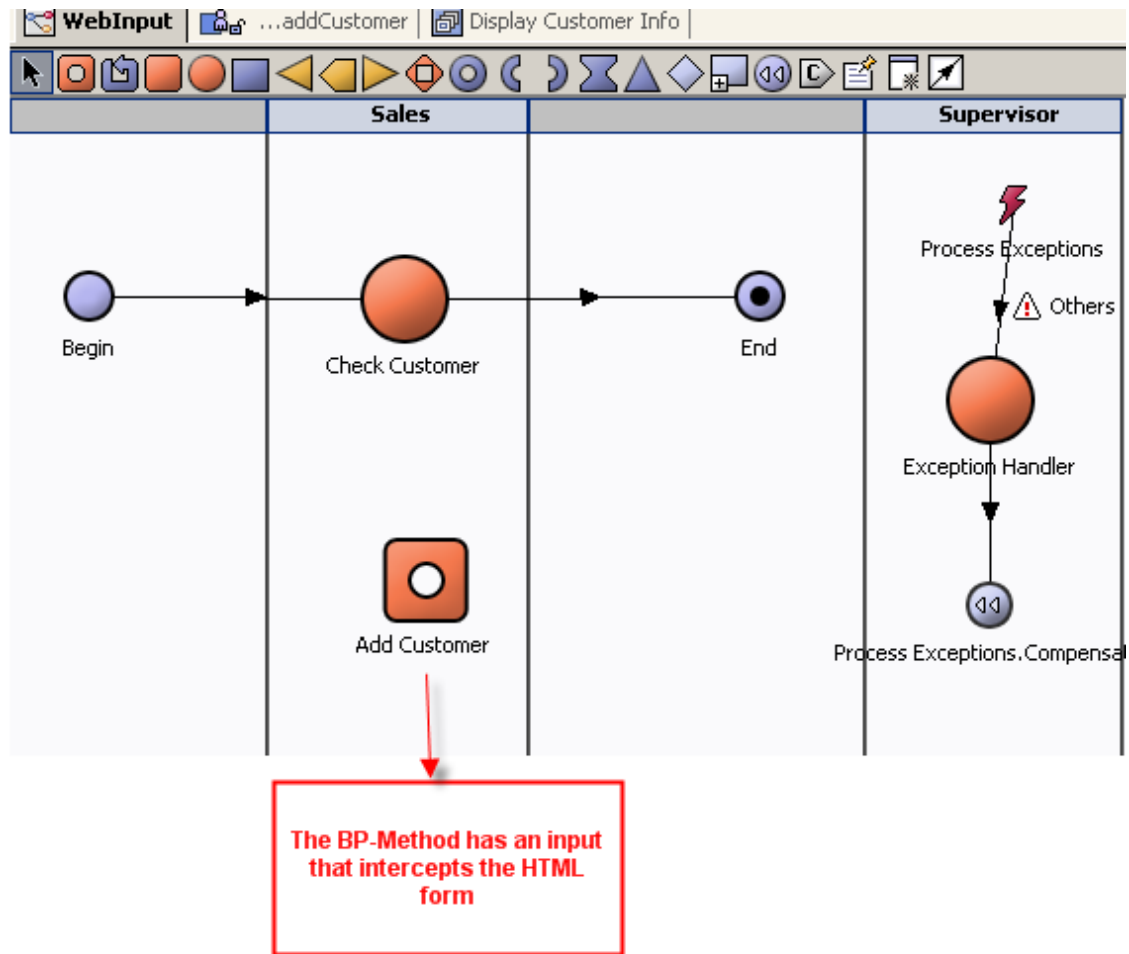
```
//Order is a Fuego Object with a presentation
//called 'auditView'
input order
  using
    selectedPresentation="auditView"
```

Input Examples

Using Interceptor Input

Intercepting a HTML Form

Customer's information is captured in an HTML form. For each completed form, an instance is created in the process.



The HTML source of the Web page is as follows:

```
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type"
      content="text/html; charset=iso-8859-1">
</head>
<body bgcolor="#FFFFFF">
<form method="post" action="">
<p>Enter your customer number:
<input type="text" name="customerNumber">
</p>
<p>Enter your name:
<input type="text" name="customerName">
</p>
```

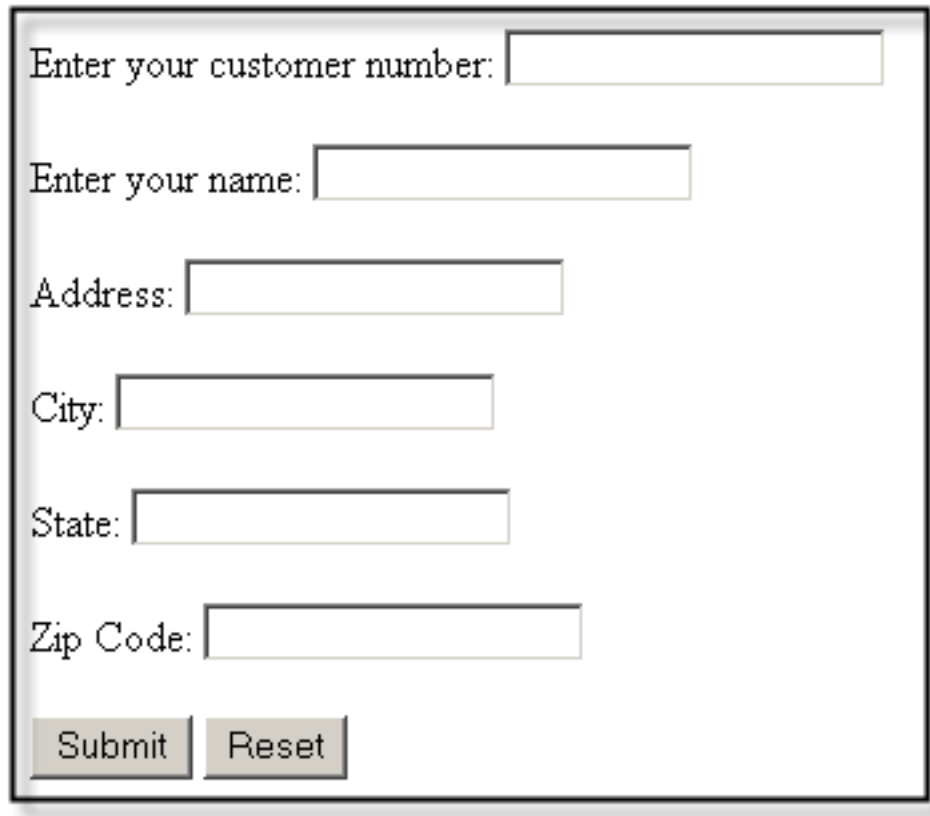


```
<p>Address:
<input type="text" name="address">
</p>
<p>City:
<input type="text" name="city">
</p>
<p>State:
<input type="text" name="state">
</p>
<p>Zip Code:
<input type="text" name="zipCode">
</p>
<p>
<input type="submit" name="Submit" value="Submit">
<input type="reset" name="reset" value="Reset">
</p>
</form>
</body>
</html>
```

The BP-Method is the following:

```
input "customerNumber" customerNbrArg,
      "customerName" customerNameArg,
      "address" addressArg,
      "city" cityArg,
      "state" stateArg,
      "zipCode" zipcodeArg
using title = "Enter order information",
htmlForm =
  "http://localhost:9595/InterceptorCase01/
    TestInterceptor/classRegister.html",
links = "clear",
userControl = false
```

When executed the following form is displayed:



Enter your customer number:

Enter your name:

Address:

City:

State:

Zip Code:

Once the form is completed and you select the *Submit* button, as no *action* is defined in the html, the interception is finished and the control is returned to the BP-Method. An instance is created and flows through the process.

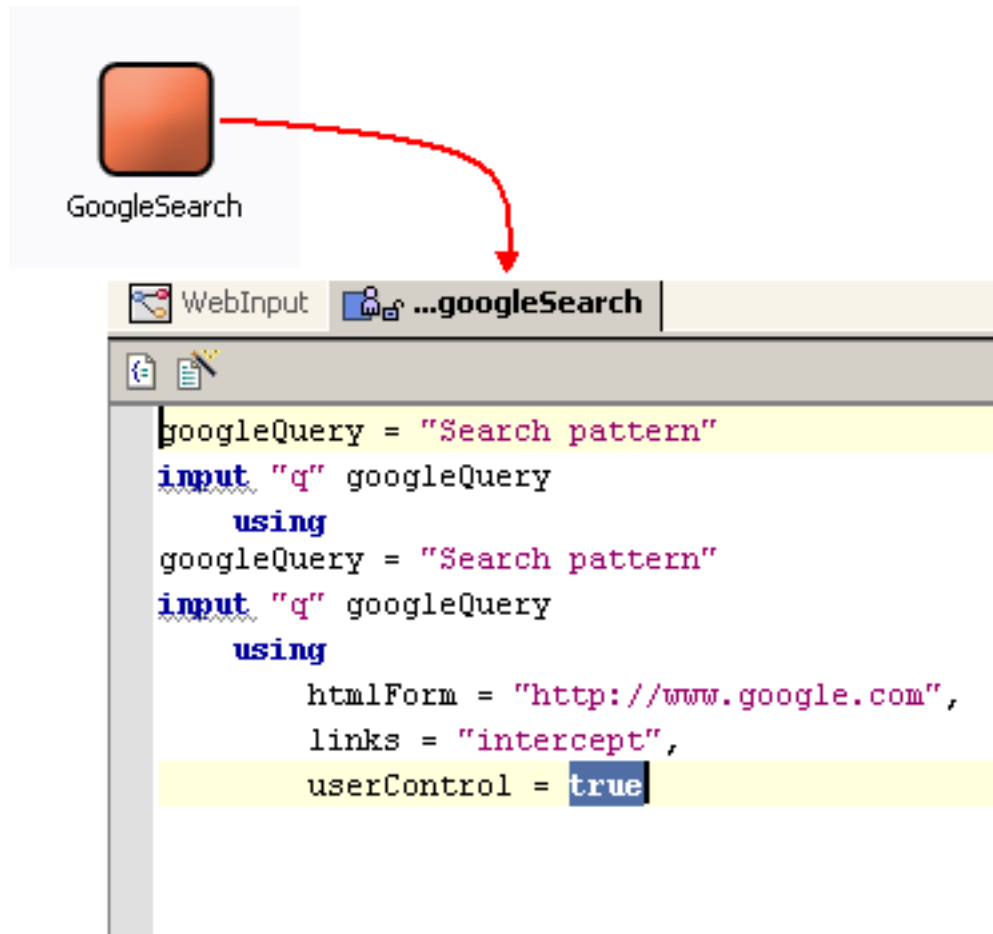
Find the project **InterceptorCase01.fpr** in FuegoBPM's installation directory, in the studio/samples directory to follow the example.

Create the **classRegister.html** file with the above html code. Copy this file into the Fuego's installation directory, in studio or enterprise/webapps/portal/TestInterceptor directory.

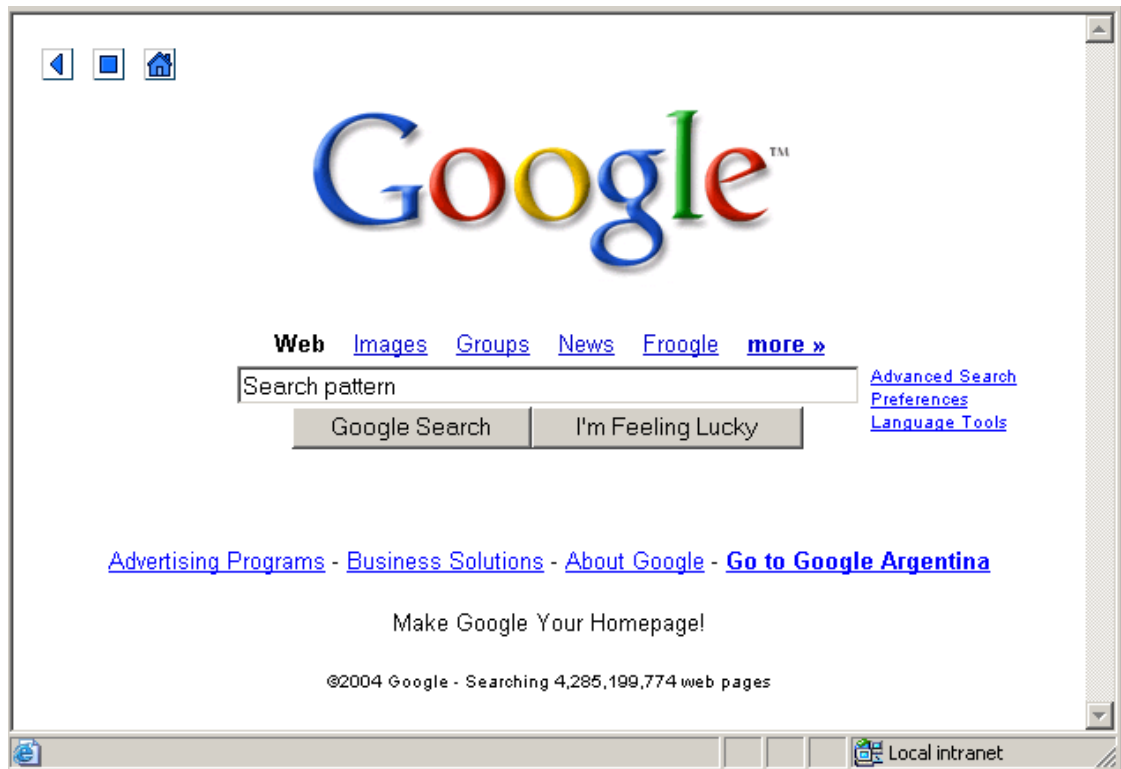
The Fuego's participant name is: test1.

Intercepting a URL page




The following is an example where a **global activity** is used to intercept a web page.



When executed, **Google's page** is displayed:



The **links** option is set to **intercept**, therefore as the intercepted page might contain links, as the user clicks on one of them, the link will be intercepted and displayed in the same window.

The **userControl** option is set to **true**, therefore the the navigational control bar    helps the user to go back, finish the interception or begin again.

Intercepting a JSP

The following process has an activity with multiple tasks.

Each task is an example on how interception can be implemented.

- Case 1: Intercepting using -> links=clear, until = blank
- Case 2: Intercepting using -> links=clear, until != Filename&FullURL

- Case 3: Intercepting using -> links=clear, until = Filename
- Case 4: Intercepting using -> links=popup, until = blank
- Case 5: Intercepting using -> links=intercept, until = blank
- Case 6: Intercepting using -> links=popup, until = Filename&FullURL
- Case 7: Intercepting using -> links=popup, until = Filename
- Case 8: Intercepting using -> links=intercept, until = Filename&FullURL
- Case 9: Intercepting using -> links=intercept, until = Filename
- Case 10: Intercepting using -> links=clear, until = blank, readonly&required
- Case 11: Intercepting using -> links=intercept, until != pattern@Filename
- Case 12: Intercepting using -> links=clear, until = pattern@Filename
- Case 13: Intercepting using -> links=popup, until = pattern@Filename

Find the project **InterceptorCase02.fpr** in FuegoBPM's installation directory, in the studio/samples directory to follow the example.

The example uses the files **InterceptorTestCaseStep1.jsp**, **InterceptorTestCaseStep2.jsp** and **InterceptorTestCaseStep3.jsp** located in the project's root directory.

The FuegoBPM's participant name is: test1.

Long running statements

To protect the FuegoBPM Engine against Method tasks that are not behaving as expected (such as Method with an infinite loop) and remote components, every Method has a *timeout* property that controls the maximum duration of Method execution. By default, this property is set to a five-minute ('5m') interval, which starts counting from the moment the Method begins its execution.

```
i = 1
while i > 0
    // ... code here
    i = i + 1
end
```

If you run the Method as displayed above without some protection mechanism, the Method would run forever (or until the engine is shut down). Besides locking the engine, infinite Method tasks also lock the instance so that no other user is able to process it. To keep locking from occurring, the *timeout* property invokes and ends the Method in five minutes.

The *timeout* property is checked when the following conditions occur:

- Method enters a loop or iteration.
- A remote component is invoked.

Changing Method timeouts

To change a Method timeout:

```
// Set the maximum time the Method can run
timeout = '10m'

// Execute a loop or iteration for 10 minutes at most
```

```
for each i in myArray do
  // statements
end
```

If the Method task is in an Interactive activity, consider using "relay to" . The *relay to* statement automatically ends Method execution for the activity when you expect that a user will take a certain amount of time to finish the execution. When the user finally responds, the response is routed to an alternate method designated in the *relay to* statement.

Relay to statements

Note



From this version onwards, consider using Screenflows instead of "relay to" invocations since Screenflows have all the benefits of **relay to** invocations with the simplicity of process-like design.

Controlling long running statements with *relay to*

When developing Method, consider the impact the code will have on your process. For example, code in Interactive activities often requires some kind of response from a human end user.

By nature, humans are unpredictable. A human end user may start his or her task in Work Portal but then something may cause him or her to forget to complete the task until a later time.

Uncompleted tasks lock resources in the Engine. Locked resources not only lock the method, but also decrease the scalability of the engine, which means that the engine cannot accommodate as many users as it can do under normal circumstances. To ensure that resources do not lock while waiting for end user input, you can use the *relay to* in your code.

When you use *relay to* in an Interactive method, the engine

immediately ends the execution of the method and does not wait for end user input. This frees the resources to handle new instances. When the end user finally enters his or her input, the engine routes the instance with the end user input to the method designated in the *relay to*.

Relay to example

In the following example, a simple input is requested. The *relay to* statement is then invoked.

```
input "Enter something in the box here: " name
  using title = "Relay To Example"
  relay to CilReachedFromRelayTo
    using relayToName = name
```

The method in the *CilReachedFromRelayTo* task is as follows:

```
// This is the standalone Method reached by an input
// statement in some Method in the process.
// Usually, the only Method you see in it is setting
// instance variable(s) from the Method's incoming
// argument variable(s).

name = arg.relayToName

display "This is the standalone Method reached from another
  Method's input statement with a \"relay to\".
  You entered: " + name
```

For further information, refer to Using Relay To.

Control Statements

Control statements enable the program to make a decision about what to do next. Fuego Business Language (FBL) provides several control statements:

- Compound Statements: Group other statements.
- Conditional Statements: Change the execution sequence based on a condition.
- Multi Path Conditional Statements: Change the execution sequence based on several conditions.
- Bounded Loops: Loop with a known bound.
- Unbounded Loops: Loop with an unknown bound (checked by a condition).
- Exit Statements: Stop the execution of the current control statement.
- Labeled Statements: Statements with labels.
- Throw Statements: Signal an exception.

Compound Statements

A compound statement groups other statements in a logical unit. It defines a scope and allows you to handle exceptions and execute statements that **must** always be executed regardless of the outcome of the block.

Syntax

The most basic form of a Compound Statement simply encloses some statements together:

```
do
    //Your statements here
end
```

You can also handle exceptions that are thrown inside the block:

```
do
  //Your statements here
on ex as Java.Exception
  //Handle Exception here
end
```

You can add some code to be executed when your block finishes, regardless of whether it has finished by exception or not:

```
do
  //Your statements here
on exit
  //Do something that must be done always, such as
  //releasing external resources
end
```

Let's see an example combining all the features:

```
do
  //Your statements here
on ex as Java.Exception
  //Handle Exception here
on exit
  //Do something that must be done always, such as
  //releasing external resources
end
```

About Loops and Methods

Loops and methods define a block, which can be used to handle exceptions.

Loops

Loops define an enclosing compound statement, which encloses the whole loop, as in this example:

```
names as String[]
names = ["John", "Peter", "Mary"]
for each name in names do
    // Do something
on ex as Java.Exception
    // Handle your exceptions
end
```

which is equivalent to:

```
names as String[]
names = ["John", "Peter", "Mary"]
do
    for each name in names do
        // Do something
    end
on ex as Java.Exception
    // Handle your exceptions
end
```

Methods

Methods define a compound statement that contains the entire body of the method. This block is labeled after the method's name, so you can add exit and exception handlers directly in the method:

```
    //Your statements here
on ex as Java.Exception
    //Handle Exception here
on exit
    //Do something that must be done always, such as
    //releasing external resources
```

Conditional Statement

This statement allows the conditional execution of a statement or group of statements based on one or more logical conditions.

Syntax

```
if <condition> then
    statements
end
```

Examples

The following example is used with *display* and *input* statements to capture end user feedback. This particular example evaluates the variable *selected* and sets the predefined *action* variable to FAIL.

```
if selected = "Cancel" then
    action = FAIL
end
```

The next example evaluates the variable *orderTotal*. If the order is greater than \$5,000, the Boolean variable *checkCredit* is set to *true*.

```
if orderTotal > 5000 then
    checkCredit = true
end
```

We can go one step further with the previous example. We can also check whether the order is a credit order or a cash order by using a logical operator.

```
if orderTotal > 5000 and paymentType = "Credit" then
    checkCredit = true
end
```

The final example shows another use of a logical operator. This

example checks whether the variable *lollipop* is *cherry* or *raspberry*. If so, *eat* is set to *true*.

```
if lollipop = "cherry" or lollipop = "raspberry" then
    eat = true
end
```

Deciding between two courses of action

This statement is similar to the simple conditional statement. They evaluate a Boolean expression and execute a specific expression if *true*. It executes a different expression if *false*. These statements provide advanced decision-making power in your program.

Syntax

```
if <condition> then
    statements
else
    statements
end
```

```
// if elseif
if <condition> then
    statements
elseif <condition> then
    statements
elseif <condition> then
    statements
end
```

Examples

The following example is a common "if else" statement that you typically use with display and input statements. It evaluates the variable *selected*, which has captured user input and sets the predefined variable *action* to FAIL if true, or sets some instance variables if false.

```
if selected = "Cancel" then
    action = FAIL
else
    orderStatus = "Reviewed"
end
```

The next example takes the previous example one step further by using another *elseif* statement. This way, you can continue adding conditions indefinitely.

```
if selected = "Cancel" then
    action = FAIL
elseif selected = "Process" then
    orderStatus = "Reviewed"
    financeStatus = "Check"
else
    orderStatus = "In Review"
end
```

Multi Path Conditional Statement

This statement is basically an enhanced version of the "if-else" statement. It is more convenient to use when you have code that needs to choose one path among many to follow.

Syntax

```
case <expression>
when <cases1> then
```

```
        [statements]
when <cases2> then
    [statements]

...

when <casesN> then
    [statements]
else
    [statements]
end
```

- Cases are comma delimited expressions.
- There could be zero or more case clauses.
- The 'else' section is optional.

This case expression is then matched against the value following each "when", and if there is a match, it executes the code contained inside that block. If no match is found, it executes the else block at the end of the statement.

Example

```
case dayNumber
when 1 then
    shortDesc = "Sun"
when 2 then
    shortDesc = "Mon"
when 3 then
    shortDesc = "Tue"
when 4 then
    shortDesc = "Wed"
when 5 then
    shortDesc = "Thu"
when 6 then
    shortDesc = "Fri"
when 7 then
```

```
        shortDesc = "Sat"
else
    throw InvalidDayNumber(dayNumber)
end
```

Bounded Loop

Bounded loops allow you to execute a set of statements a known number of times. The number of times might be determined by a range or a collection.


Range iteration

This loop iterates over an integer range. That is, on each iteration, an integer variable increments by one.

Syntax

```
for <id> in <rangeStart> .. <rangeEnd> do
    <statements>
end
```

Note

 *rangeStart* and *rangeEnd* can be an expression. The range limits are inclusive. Modifying the variable inside the loop does not have any effect on the loop's execution.

Example

This example displays the numbers from 1 to 3.

```
for i in 1..3 do
    display i
end
```



Key iteration

Allows you to iterate over the keys or indexes of an array.

Syntax

```
for <id> in <expression> do
  <statements>
end
```

Note

 * The *expression* must yield an array.

Example

This example displays all the keys in the *ages* array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for name in ages do
  display name
end
```

Element iteration

Iterates over the values of an array. Some of them may be excluded by adding a "where" clause.

Syntax

```
for each <id> in <expression> [where <condition> ] do
  <statements>
end
```

Note

* The *expression* must yield an array.

Examples

This example displays all the values in the *ages* array:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for each age in ages do
  display age
end
```

The following example shows only the ages above 25:

```
ages as Int[String]
ages = ["John" : 23, "Peter" : 42, "Mary" : 29]

for each age in ages
  where age > 25
do
  display age
end
```

Stopping a Loop

Loop execution can be stopped by using an Exit Statement.

Unbounded Loop

Unbounded loops are useful when the programmer does not know in advance how many times the loop will be traversed.

This loop repeats an action until an associated test condition returns false or until an exit statement is executed. The condition is evaluated before entering the loop and after each iteration.

Syntax

```
while <condition> do
  <statements>
end
```

Example

Continue asking until the user chooses "Ok":

```
selection as String = ""
while selection /= "Ok" do
  display "Are you sure ?"
  using buttons = ["Ok", "Cancel"]
  returning selection = selectedButton
end
```

Exit statement

A method or a loop in a method can be interrupted by using the *exit* statement:

- If the loop is labeled and the exit statement refers to that label, the exit statement exits the labeled loop.
- If the provided name is the method name, the execution of that method ends.
- If no name is provided, execution exits the innermost loop of the method, or the method itself, if it is outside a loop.

Syntax

```
exit [label] [when <condition>]
```

Note



label can be the method name or a labeled loop. The *condition* can be used to avoid cluttering your code with a conditional statement.

Examples

The following example finds the first e-mail containing a specific subject then exits the loop.

```
// order is a variable of type Fuego.Net.Mail
order as Mail
url   as String = ""

for each mail in MailServer(url, false).messages do
  if mail.subject = "New Order" then
    order = mail
    exit
  end
end

// if there is no order to process, stop method execution
exit when order is null
```

The second example finds a participant, assigns it to the *nextParticipant* variable, and ends the method execution. The name of the method is *findParticipant*.

```
participantName = "John"

for each p in activity.role.participants do
  if p.name = participantName then
    // participant found!
```

```
        nextParticipant = p
        exit : findParticipant
            // findParticipant is the name of the method
    end
end
```

For further information refer to sections:

- Compound Statements
- Bounded Loops
- Labeled Statements

Labeled Statements

Labels provide a name to identify and reference a statement. Labels are used by the exit statements to specify the statement to which the exit applies.

Note that all methods have an enclosing label which has the same name as the method.

Syntax

```
<label> : <statement>
```

Example

```
// order is a variable of type Fuego.Net.Mail
order as Mail
url   as String = ""
```

```
mainLoop: for each mail in
            MailServer(url, false).messages do
    if mail.subject = "New Order" then
        order = mail
        exit : mainLoop
    end
end

// if there is no order to process, stop method execution
exit when order is null
```

Throw Statement

The throw statement lets you raise an exception, thus breaking the execution until:

- The exception is handled.
- The method finishes and the process' exception handling procedures gain control.

Syntax

```
throw <expression>
```

Note



* *expression* must yield a `Java.Lang.Throwable`.

Example

```
throw Java.Exception("Something is wrong")
```

Assignment

The assignment statement allows you to evaluate an expression and store the result in a variable, in an attribute or in an array element.

Syntax

The basic syntax is as follows:

```
<variable> = <expression>
```

Do not mistake the assignment statement for the equality operator. In Fuego style, both of them use the same symbol. See the following example:

```
name as String
name = "John"
if name = "Peter" then
    //Do something...
end
```

Assignment by Value vs Assignment by Reference

You have two kinds of assignments:

- By Value
- By Reference

The kind of assignment used depends on the variable type and expression.

Assignment by value

An assignment by value copies the value of the expression to the variable. This kind of assignment is performed on the following types:

- Bool
- Int
- Real
- Decimal
- String
- Time
- Interval

Assignment by reference

An assignment by reference copies a reference to the type returned by the expression. This kind of assignment is used for all types except the ones above-mentioned. See the following example:

```
mail as Mail
other as Mail
mail.from = "someone@example.com"
other = mail
other.from = "someone_else@example.com"
display mail.from
```

The example above displays "someone_else@example.com". Note that although you modify **other.from** when **mail.from** is displayed, they both have the same value. This is because when you did:


```
other = mail
```

you simply copied the reference to Mail, not the value. So, when a member accesses any of the variables, he/she is actually having access to the same instance as that of the Mail component.

If you want to have separate instances, you must use the clone function:

```
mail as Mail
other as Mail
mail.from = "someone@example.com"
other = clone(mail)
other.from = "someone_else@example.com"
display mail.from
```

The example above shows "someone@example.com". This is because the clone function duplicates an object's instance.

Logging Statement

The logging statement is used to log a message in the log files maintained by FuegoBPM

Engine. Log messages are useful to debug the behavior of your code, especially in automatic activities.

Syntax

```
logMessage "message to log"
  [using
    [severity = <severity>]
    [, detail = <detail>]
  ]
```

Severity levels

The following are severity levels that can be used with the logging statement:

- DEBUG
- INFO
- WARNING
- SEVERE
- FATAL

You can choose the severity level you want to display in the Engine logs in the Execution Console.

Example

```
orderName = 1
customer = 1

logMessage "Order " + orderNumber + " from customer "
          + customer + "was aborted" using severity = SEVERE
```

Type conversions

Conversion between variable types can be accomplished by "forcing" a type on a variable of another type. There are two syntaxes to make the conversion: Functional syntax and the Conversion operator.

Functional syntax

The value to be converted is passed as an argument to the type

name. The following examples show you how to force a type on a variable:

- Any variable type can be converted into a String.

```
someNum as Int = 23
someString as String
someString = String(someNum)
someString = String('now')
```

- Any variable type can be converted from a String. However, this operation can fail if the format of the String is not valid for the type to which you are converting. For example, Time has certain formats that must be followed. For further information, refer to Times and Intervals.

```
localTime as Time
localTime = Time("2002/01/20 17:39:23")
```

Conversion operator

Conversions can also be used by using the Conversion operator:

```
localTime as Time
localTime = "2002/01/20 17:39:23" to Time
```

The Conversion operator is especially important when dealing with Transformations.

Transformations

Transformations allow a program to perform a conversion from the result of one or more *source expressions* to a new value of a specified *target type*. This conversion is performed by following a set of rules defined in a transformation, which is stored in a *transformation library*.

Syntax

Transformation calls are expressions with the following syntax:

```
<source expression> to <target type>  
    using <transformation library>
```

or

```
<source expression 1>, ..., <source expression N>  
    to <target type> using <transformation library>
```

- *Sources* are expressions whose result will be transformed into *target type*.
- *Target type* is any type defined in the catalog.
- *Transformation library* is the component that contains the definition of the transformation.

Chapter 13. Exceptions

Exceptions

FuegoBPM Studio allows you to use and handle exceptions in your methods and processes.

For further information about handling exceptions:

- in your code
 - Compound Statement
 - Throw Statement
- in your processes
 - Process Exception Flow
 - Exception Transition
 - Exception Handling

The following list includes some of the most common Java exceptions that can appear:

- ClassNotFoundException
- NumberFormatException
- NegativeArraySizeException
- ArithmeticException

- CloneNotSupportedException
- NullPointerException
- ArrayStoreException
- IllegalThreadStateException
- StringIndexOutOfBoundsException
- Exception
- ArrayIndexOutOfBoundsException
- IllegalAccessException
- UnsupportedOperationException
- InterruptedException
- InstantiationException
- IllegalArgumentException
- IndexOutOfBoundsException
- RuntimeException
- SecurityException
- IllegalMonitorStateException
- ClassCastException
- IllegalStateException

ClassNotFoundException

The *ClassNotFoundException* is thrown when a required class is not found at run-time. This may happen because a jar file is missing.

NumberFormatException

The *NumberFormatException* is thrown to indicate that the application has attempted to convert a string into one of the numeric types, but the string does not have the appropriate format.

NegativeArraySizeException

The *NegativeArraySizeException* is thrown if an application tries to create an array with negative size. Since arrays can be empty, but not negative, this exception is thrown.

ArithmeticException

The *ArithmeticException* is thrown when an exceptional arithmetic condition has occurred. For example, a division by zero on integer numbers (Real numbers support division by zero and return NaN).

The following example shows the use of a Compound Statement to display the exact arithmetic exception that has occurred:

```
do
  divisor as Int = 0
  dividend as Int = 26
  result as Int
  result = dividend / divisor
on e as Java.ArithmeticException
  display e.message
end
```

CloneNotSupportedException

The *CloneNotSupportedException* is thrown to indicate that the *clone* method has been called to clone an object but the object's class does not implement the *Cloneable* interface. In other words, the object could not be cloned.

NullPointerException

The *NullPointerException* is thrown when an application attempts to use **null** in a situation in which an object is required.

ArrayStoreException

The *ArrayStoreException* is thrown when you attempt to store the wrong type of object in a native array (certain arrays returned by an introspected Java component). For example, this is thrown if you try to store a decimal variable into a String-type array. This is similar to a *ClassCastException* but for array elements.

IllegalThreadStateException

The *IllegalThreadStateException* is thrown to indicate that a thread is not in an appropriate state for the requested operation.

StringIndexOutOfBoundsException

The *StringIndexOutOfBoundsException* is thrown when accessing a character inside a string, and the requested index is out of bounds.

Exception

The *Exception* exception and its subclasses are a kind of *Throwable* and indicate conditions that a reasonable application might want to catch. Basically, *Exception* is an extension of *Throwable*, which extends *RuntimeExceptions*.

ArrayIndexOutOfBoundsException

The *ArrayIndexOutOfBoundsException* is thrown when you try to access an array with an illegal index number, e.g.:

```
names as String[]
```



```
names = ["John", "Peter", "Mary"]  
display names[3]
```

IllegalAccessException

The *IllegalAccessException* is thrown when an application tries to access a class or a method, but the currently executing method does not have enough privileges to do so because the class or method is not public or it is in another package.

This exception might happen if you change the access flags of an introspected component and replace the jar without re-introspecting it.

UnsupportedOperationException

The *UnsupportedOperationException* is thrown when you have requested an operation that is not supported.

InterruptedException

The *InterruptedException* is thrown when a thread is waiting, sleeping or otherwise paused for a long time and another thread interrupts it using the interrupt method in Thread class.

InstantiationException

The *InstantiationException* is thrown when an instance of a certain class cannot be created.

IllegalArgumentException

The *IllegalArgumentException* is thrown to indicate an argument passed to a method is illegal.

That could mean that:

- an argument is out of range
- a certain argument is null but it is required

IndexOutOfBoundsException

The *IndexOutOfBoundsException* is thrown to indicate that an index is out of bounds.

RuntimeException

The *RuntimeException* is the superclass of all exceptions that may happen at runtime because of some unexpected condition.

The following exceptions are subclasses of *RuntimeException*:

- *ArithmeticException*
- *IndexOutOfBoundsException*
- *NegativeArraySizeException*
- *NullPointerException*
- *ArrayStoreException*
- *ClassCastException*
- *IllegalArgumentException*
- *SecurityException*
- *IllegalMonitorStateException*
- *IllegalStateException*

- `UnsupportedOperationException`

SecurityException

The *SecurityException* is thrown by the security manager to indicate some kind of security violation. For example if a method attempts to exit the Java Virtual Machine while running at the server, a security exception will be thrown.

IllegalMonitorStateException

The *IllegalMonitorStateException* is thrown to indicate that a thread has attempted to wait on an object's monitor or to notify other threads waiting on an object's monitor without owning the specified monitor.

ClassCastException

The *ClassCastException* is thrown when you try to store a value into a variable and the type of the value cannot be converted to the type of the variable. This may happen in several situations:

- when narrowing a type using a forced conversion.
- when you try to interface the process with another process or with an external application and you pass the wrong type of variable. For example, if a subprocess expects variables of string type and your process sends variables of decimal type, there is a type conflict.

IllegalStateException

The *IllegalStateException* is thrown when an attempt to call a method at a time when it is illegal to do so.

Chapter 14. Embedded SQL

Embedded SQL

FuegoBPM Studio supports embedded SQL written directly in the code (ANSI-92 Entry Level). This section describes the supported syntax and provides some examples.

For further information about introspecting SQL components, refer to SQL Components.

INSERT statements

INSERT adds one or more row(s) to a table. The values order must match the order of the corresponding attributes at table creation time. Otherwise, the column names corresponding to the values must be specified in the INSERT statement. Columns not specified in the column list when they are explicitly defined are set to their default values or to null.

Note



In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Syntax

```
INSERT INTO <table_name> [(column_name1, col_name2,...)]  
VALUES (value1, value2,...);
```

or

```
INSERT INTO <table_name> [(col_name1, col_name2,...)]  
<select statement>;
```

Example

```
INSERT INTO employee(fname, lname, salary)  
VALUES ("John", "Smith", 25000);
```

Note that the values can also be an expression:

```
firstname = "John",  
salary = 20000  
  
INSERT INTO employee(fname, lname, salary)  
VALUES(firstname, "Smith", salary + 5000);
```


Example using INSERT with a SELECT statement

```
INSERT INTO student  
SELECT *  
FROM employee  
WHERE salary > 30000;
```

DELETE statements

DELETE removes all rows that satisfy a given condition from a table.

Note

 In the syntax and example sections below, certain words appear in all capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Syntax

```
DELETE FROM <table_name>  
[WHERE condition];
```

Example


Deletes all employees whose first name is "John" and its last name is "Smith":

```
DELETE FROM employee  
WHERE fname = "John" and lname = "Smith";
```

UPDATE statements

The UPDATE statement modifies a set of row values that satisfy a given condition.

Note

 In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Syntax

```
UPDATE <table_name>
```

```
SET column_name1 = value-expression1,
    column_name2 = value-expression2,
    ...
[ WHERE condition ];
```

Example

Increases the salary of all the employees who earn less than \$25,000 by 10%:

```
UPDATE employee
SET salary = salary * 1.1
WHERE salary < 25000;
```

Note



If the *where* condition is not specified, all the rows will be updated.

SELECT statements

The SELECT statement finds and retrieves a set of rows from one or more tables of a database, given a certain criteria.

Note



In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Syntax

```
for each <variable> in
    SELECT [DISTINCT | ALL] column1, column2,...
    FROM table1, table2, ...
```

```

[WHERE condition]
[GROUP BY grouping-column1, grouping-column2,...]
[HAVING group-selection-condition1]]
[ORDER BY ordering-col1 [ASC | DESC],
              ordering-col2 [ASC | DESC],...]
do
    // ....
end

```

A column can be a column name or an expression that can include an aggregate function and can be renamed using *column AS newName*. An ordering column can also be a number *n* corresponding to the *nth* column in the column list.

Examples

```

for each e in
    SELECT *
    FROM employee
    WHERE salary > 25000
    ORDER BY lname, fname
do
    display "employee name: " + e.lname +
            ", " + e.fname
end

```

```

for each e in
    SELECT depnumber, COUNT(*), AVG(salary)
    FROM employee
    WHERE depnumber != 3 and depnumber !=4
    GROUP BY depnumber
    HAVING COUNT(*) > 5
    ORDER BY depnumber
do
    // ...
end

```



Using aggregate functions

The following example selects the maximum salary in the employee table using the MAX function. The *row.1* term is used to specify the first column. The variable *salary* is used to store the result of the maximum salary in the table.

```
for each row in
    SELECT MAX(salary)
    FROM employee
do
    salary = row.1
end
```

Referencing columns

Note

 In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Examples

Defining columns explicitly

The values of a row can be accessed by name or by position if the columns are explicitly defined.

```
for each e in
    SELECT fname, lname, salary
    FROM employee
do
    // accessing by name
    firstName = e.fname
    lastName = e.lname
    salary = e.salary
```

```
// or accessing by position
firstName = e.1
lastName = e.2
salary = e.3
end
```

Note



Reference by position cannot be used when the columns are specified as '*'.
as '*'.

Using the AS clause

When an expression or an aggregate function appears in the columns, reference must be made using the column's position or the column must be named using the *AS* clause.

```
for each e in
  SELECT "department #" || depnumber AS dep,
        COUNT(*) AS c,
        AVG(Salary) * 1.10 AS avgsalary
  FROM employee
  GROUP BY depnumber
do
  department = e.1
  count = e.2
  avgSalary = e.3

  // or
  department = e.dep
  count = e.c
  avgSalary = e.avgsalary
end
```

Dealing with ambiguous names

When more than one column has the same name, explicit qualification or renaming is required.


Given these two tables:

```
EMPLOYEE (name      VARCHAR (20),
           depnumber INTEGER)

DEPARTMENT (depnumber INTEGER,
            name      VARCHAR (20))
```

```
for each emp in
    SELECT employee.name, department.depnumber,
           department.name AS depname
    FROM employee, department
    WHERE employee.depnumber = department.depnumber
do
    display "Employee name : " + emp.name +
           ", department #" + emp.depnumber +
           ", department name : " + emp.depname
end
```


Note

 Columns *depnumber* and *name* must be fully qualified because they appear in both tables.

Agregate functions

Aggregate functions act on all records in a query, counting rows, averaging fields, and so forth. These functions can only be used in the column section of a SELECT statement.

Note

 In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Syntax

```

AVG      ( [ ALL | DISTINCT ] column_name )
MAX      ( [ ALL | DISTINCT ] column_name )
MIN      ( [ ALL | DISTINCT ] column_name )
SUM      ( [ ALL | DISTINCT ] column_name )
COUNT ( [ ALL | DISTINCT ] column_name ) or COUNT(*)

```

Function	Description
COUNT	Counts the values of a column.
AVG	Average of the values of a column.
SUM	Sum of the values of a column.
MAX	Maximum values of a column.
MIN	Minimum values of a column.

These functions accept a single column name as an argument with the exception of COUNT, which also accepts '*'. DISTINCT or ALL can be used with any of the functions.


Example:

```

for each e in
    SELECT COUNT(*) AS c1, COUNT(DISTINCT salary) AS c2,
    AVG(salary) AS savg, SUM(salary) AS ssum,
    MAX(salary) AS smax, MIN(salary) AS smin
    FROM employee
    WHERE salary > 25000
do
    display "count of rows is " + e.c1
    display "count of distinct salary values is " + e.c2
    display "avg is " + e.savg
    display "sum is " + e.ssum
    display "max is " + e.smax
    display "min is " + e.smin
end


```

Note

 The AS clause specifies a name for the variable resulting from the use of the aggregate function.

Positioned updates and deletes

Note

 In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

Positioned update

A positioned update is performed invoking the *store* method over the current row in a *for each* statement after changing the row values.

Example

```
for each emp in
  SELECT *
  FROM employee
  WHERE depnumber = 5
do
  display "Increase salary of: " + emp.fname + "?"
    using options = ["Yes", "No"]
    returning selectedButton = selection

  if selectedButton = "Yes" then
    emp.salary = emp.salary * 1.10
    store emp
  end
end
```

Positioned delete

Positioned delete is performed by invoking the *remove* method over the current row in a *for each* statement.

Example


```
for each emp in
    SELECT *
    FROM employee
    WHERE depnumber = 5
do
    display "Remove employee : " + emp.fname + "?"
        using options = ["Yes", "No"]
        returning selectedButton = selection

    if selectedButton = "Yes" then
        remove emp // remove current employee
    end
end
```

SQL operators

SQL operators allow you to manipulate data found in SQL databases. The following operators are supported:

Note

 In the syntax and example sections below, certain words appear all in capital letters. This is an SQL convention and it is not necessary in FuegoBPM Studio. However, it is a useful convention to differentiate SQL commands from regular code.

LIKE operator

The LIKE operator searches for strings that match a specific pattern. The character (percent) ' % ' is used to match any number of characters and the underscore ' _ ' is used to match one character.

Example

```
for each e in
    SELECT *
    FROM employee
    WHERE fname LIKE "J%"
```

```
do
    // do something here
end
```


Concat operator (||)

In SQL statements, the || operator is used to concatenate two values of any type.

Example

```
for each e in
    SELECT lname || ", " || fname AS fullname
    FROM employee
do
    display "full name: " + e.fullname
end
```

Note

 In Fuego and Java style, the || symbol means "or" and it is used in conditional expressions. For further information, please see Logical operators .

IN operator

The IN operator selects values that appear in another set of values.

Syntax

```
column_name IN (value1, value2,...)
```

Example

```
for each e in
  SELECT lname, fname
  FROM employee
  WHERE salary IN (20000, 25000, 30000)
do
  display "name: " + e.lname
end
```

This statement is equivalent to the following:

```
for each e in
  SELECT lname, fname
  FROM employee
  WHERE salary = 20000 or salary = 25000 or salary = 30000
do
  display "name: " + e.lname
end
```

IS operator

The IS operator locates a record that does or does not have a null value for a particular column.

Syntax

```
column_name IS [NOT] NULL
```

Example

```
for each e in
  SELECT lname, fname
  FROM employee
  WHERE address IS NOT NULL
do
  display "name: " + e.lname + ", address: " +
```



```
end          e.address
```

BETWEEN Operator

The BETWEEN operator allows you to select records that are between two values.

Syntax

```
<column_name> [NOT] BETWEEN value1 AND value2
```

The expression *a BETWEEN b AND c* is equivalent to *a >= b AND a <= c* .

Example

```
for each e in
  SELECT lname, fname
  FROM employee
  WHERE salary BETWEEN 20000 AND 30000
do
  display "name: " + e.name + ", salary: "
    + e.salary
end
```

SQL keywords

Some words in a SQL statement have a special meaning and cannot be used as regular identifiers. These keywords include the following:

FROM, SELECT, AS, ORDER, BY, LIKE, HAVING, GROUP, ASC, DESC, DISTINCT, BETWEEN, AVG, MAX, MIN, SUM, COUNT, INTO, VALUES, ALL

Note

These keywords can be used as regular identifiers outside SQL statements. These keywords are case-insensitive inside SQL statements.

Stored procedures

Any procedure that you have developed in your database system (such as Oracle or MS SQL) is added to the catalog during introspection. The stored procedure can be treated as a method and used in your code.

Note

Any procedure that uses vendor specific features, such as *rowtype* in Oracle, is not supported. Only standard SQL procedures are added to the catalog.

Chapter 15. Regular Expressions

Regular Expressions

A regular expression is a small, very specialized, powerful language for string pattern matching. Regular expressions allow you to search strings, extract desired substrings, perform advanced searches, and replace operations on strings.

FuegoBPM provides support for regular expressions and the syntax is compatible with the Perl syntax.

Regular Expression Syntax

In FuegoBPM Studio, you write regular expressions between a single quote and a slash, and a slash and a single quote as follows:

```
'/ {regular expression} /'
```

The simplest regular expression is a single word to search for in a string. A regular expression consisting of a word matches any string that contains that word. So, the following regular expression matches any string that contains the word "hello".

```
'/hello/'
```

Functions

In FuegoBPM Studio, you use regular expressions by calling functions (methods) used on string objects. The following string functions support regular expressions:

Function	Purpose	Returns
contains()	Matches a substring contained in the string.	Bool
isMatch()	Matches the string completely.	Bool
indexOf()	Gets the first index location where the regular expression matches the string.	Int
lastIndexOf()	Gets the last index where the regular expression matches the string.	Int
match()	Attempts to match and return the substring(s) that matched the regular expression. This is useful for extraction and parsing.	String[] - The array of subexpressions (groupings) matched. When the g modifier is used, the array of matched occurrences is returned instead.
split()	Splits a string using the given regular expression as a separator.	String[] - The array of fields (pieces of the string that were separated by the given separator.)
count()	Gets the number of substrings (within the string) that the given regular expression matches.	Int
replace()	Replaces pieces of the string with new strings.	String - the new modified string.

Simple Matching

In order to find out if a particular string contains a word, you need to use the `contains()` function. For example:

```
myString.contains('/Hello/')
```

The previous line of code will search for the word "Hello" in the string that is contained in the `myString` variable. It returns `true` if "Hello" is found and `false` if it is not.

Note



You can also apply function calls to literal strings instead of variables.

For example,

```
"Hello world!".contains('/Hello/')
```

returns `true` because "Hello world!" contains "Hello".

Modifiers

You may have already noticed that matching is case sensitive. This means that the regular expression will only match a substring if both the regular expression and the substring have the same upper/lowercase characters.

In order to make matching case insensitive, you need to use a modifier. Regular expression modifiers allow you to change the default behavior of the matching.

To add a modifier, you extend the basic syntax of the regular expression as follows:

```
'/regular expression/modifier(s)'
```

Each modifier is just a single character that is specified between the last slash and the quote. The modifier for insensitive case matching is `i`. So, the follow regular expression matches all Hello, hello, HeLIO, hELLO...

```
'/hello/i'
```

Some other modifiers are listed in the following table:

Modifier	Meaning
i	Searches in a case insensitive manner.
g	Matches the regular expression as many times as possible. Matches all substrings globally.
s	Treats the string as a single line.
m	Treats the string as multiple lines.

More than one modifier can be used at the same time.

Metacharacters and Characters Sets

So far, you have only used regular expressions to search for literal words. The power of regular expressions is based on the fact that they can contain special characters that perform special functions. These characters are not treated as usual characters and are not matched literally. They are known as metacharacters.

Some of the metacharacters that make pattern matching more

generic are as follows:

[] . () * ^ \$? \

The [and] characters are used to specify a set of characters that you wish to match. Characters can be listed individually or a range of characters can be indicated by listing two characters separated by a dash (-). For example, [aeiou] matches any of the vowels. The set [a-d] is equivalent to [abcd].

If you specify the ^ character right after the opening bracket, it matches the complement of the set. In other words, any character that is not in the set. For example, [^0-9] matches any non-digit character.

The following table lists some examples of regular expressions using sets:

Regular Expression	Strings that Match
'/[cr]at/'	cat, rat
'/[0-9]/'	Any digit
'/[0123456789]/'	Any digit
'/[A-Z]/'	Any uppercase letter
'/[0-9][0-9]/'	Any two digits together (like 01, 42, 27...)
'/[aeiou]/'	Any of the vowels

Fortunately, there are some shortcuts for some of the common character sets. For example, \d denotes "any digit" the same way that [0-9] does. \D means "any non-digit" like [^0-9]. The following table lists some other common shortcuts.

Shortcut Sequence	Equivalent to
\d	[0-9]
\D	[^0-9]

Shortcut Sequence	Equivalent to
<code>\w</code>	<code>[a-zA-Z0-9_]</code> Any alphanumeric character including the underscore.
<code>\W</code>	<code>[^a-zA-Z0-9_]</code> Any non-alphanumeric character.
<code>\s</code>	Any whitespace character.
<code>\S</code>	Any non-whitespace character.

These shortcuts can be included inside a character class (set.) For example, `[\da-fA-F]` is a character class that will match one hexadecimal digit. The tab, new line, and return characters are specified with `\\t`, `\\n` and `\\r`, respectively.

Another special metacharacter is the dot (`.`). A dot within a regular expression matches any character (except the `\\n` character, unless the `s` modifier is used).

Special Cases

What happens if you want to search for some of the metacharacters, like `[` or `.`? You can escape these characters with a backslash (`\\`) just before the character. For example, `\\.`, `\\`, `\\[` match a literal dot, backslash, and opening bracket, respectively.

Go back to the Method example and try these metacharacters with different regular expressions. Experiment with character classes and the shortcuts available. The only way to learn to use regular expressions is to build some.

Matching repetitions


In the previous examples, you were only matching expressions consisting of a few generic characters and literal words. To help write more expressive patterns, you use a quantifier metacharacter. The quantifiers are as follows:

? * + { }

The quantifiers allow you to determine the number of repeats of a portion of a regular expression you consider a match. Quantifiers are located immediately after the character, character class or grouping that you want to match. The following table defines each quantifier and its meaning.

Quantifier	Definition
a?	Match 'a' one or zero times.
a*	Match 'a' zero or more times (any number of times.)
a+	Match 'a' one or more times (at least one time.)
a{n,m}	Match 'a' at least n times, but not more than m times
a{n, }	Match 'a' at least n or more times.
a{n}	Match 'a' exactly n times.

Note

 'a' in the previous table can be any character, character class or grouping. You will learn more about groupings later but basically, you can group a part of a regular expression using parenthesis.

The following are some examples using matching repetitions:

Matching repetition	Description
'\w+/'	Any alphanumeric word (one or more alphanumeric characters together.)
'/-?\d+/'	A number (one or more digits) optionally prefixed by a hyphen.
'\[a-z]+\t\d{1,5}/'	Any lowercase word, followed by a tab, followed by 1 to 5 digits

Matching repetition	Description
<code>'\w+'</code>	Any alphanumeric word (one or more alphanumeric characters together.)
<code>'/The.*dog/'</code>	Any line that is followed by anything and then dog. Examples: The nice dog The quick brown fox jumped over the lazy dog The WhateverHeredog Thedog

Now, you have enough tools to create some useful regular expressions. For example, let's build a simple regular expression to match 10 digit telephone numbers. You may start with:

```
'/d{10}/' // 10 digits (no more, no less)
```

This regular expression matches any 10 digit number but it has some weaknesses. For instance, what happens if you want to accept numbers that contain dashes (-) in between, such as 321-123-1234? Ok, you can do the following:

```
'/\d{3}-\d{3}-\d{4}/'
```

This is fine, but what if you want the dashes to be optional? Try this:

```
'/\d{3}-?\d{3}-?\d{4}/'
```

This is better, but still there are some improvements to be made. You might not want to allow a zero (0) as the first digit of the number. Thus, the first digit must be in the class [1-9] instead of \d

as follows:

```
'/[1-9]\d{2}-?\d{3}-?\d{4}/'
```

Did you understand it? Let's study it in parts:

1. First, a digit between 1 and 9: `[1-9]`
2. Next, two digits (from 0 to 9): `\d{2}`
3. Then, an optional dash: `-?`
4. Three digits: `\d{3}`
5. Another optional dash: `-?`
6. Finally, four digits: `\d{4}`

Try it in the debugger:

```
phone as String
input "Enter your phone number:" phone

if phone.isMatch('/[1-9]\d{2}-?\d{3}-?\d{4}/') then
  display "OK, a valid phone number"
else
  display "ERROR, invalid phone number"
end
```

Anchors

When you use the `contains()` function, it returns true if the regular expression matched anywhere in the string. However, sometimes, you would like to specify where in the string the regular expression should try to match. To do this, you use the anchor metacharacters.

The two most common anchor metacharacters are `^` and `$`. The `^` anchor means match at the beginning of the line and the `$` anchor means match at the end of the line. The following are examples of how they are used:

```
display "rock and roll".contains('/and/')
// displays true
display "rock and roll".contains('/~np~^~/np~and/')
// displays false
display "rock and roll".contains('/~np~^~/np~rock/')
// true
display "rock and roll".contains('/rock$/')
// false
display "rock and roll".contains('/roll$/')
// true
display "rock and roll".contains('/nd roll$/')
// true
display "rock and roll".contains('/\~np~^~/np~rock$/')
// false
display "rock and roll".contains('/\~np~^~/np~rock and roll$/')
// true
display "rock and roll".isMatch('/rock and roll/')
// true
```

The second regular expression does not match because `^` constrains `and` to match only if it is at the beginning of the string. The fifth regular expression does match, since the `$` constrains `roll` to match only at the end of the string.

Look at the last two examples. If you use both the `^` and `$`, you mean that the regular expression must match both the beginning and the end of the string. In other words, the regular expression matches the whole string. Note that both examples are equivalent since the `isMatch()` will always look for a complete match. The `^` and `$` are irrelevant when using `isMatch()`.

What is the difference between `^` and `\A` and between `$` and `\Z`?

Usually, you will only use `^` and `$`, but when using the `m` modifier, there is a small difference. If the string contains new line (`\n`)

characters, then the `^` and `$` match, just after and just before, the new line. However, the `\A` and `\Z` only match at the start and end of the whole string. So, using the `m` modifier and replacing the space between "*and roll*" by a new line you get the following:

```
display "rock and~np~\~/np~nroll".contains('/and$/m')
// true
display "rock and~np~\~/np~nroll".contains('/and~np~\~/np~Z/m')
// false
display "rock and~np~\~/np~nroll".contains('/^roll$/')
// true
display "rock and~np~\~/np~nroll".contains('/~np~\~/np~Aroll~np~\~/np')
// false
display "rock and~np~\~/np~nroll".contains('/~np~\~/np~Arock/')
// true
```

The following table outlines a more detailed explanation of the modifiers and their behavior:

Condition	Behavior
no modifier	Default behavior. <code>.</code> matches any character except <code>\n</code> . <code>^</code> only matches at the beginning of the string and <code>\$</code> only matches at the end of the string.
s modifier	Treat string as a single long line. <code>.</code> matches any character, even the <code>\n</code> . <code>^</code> only matches at the beginning of the string and <code>\$</code> only matches at the end or before a new line at the end.
m modifier	Treat string as a set of multiple lines. <code>.</code> matches any character except <code>\n</code> . <code>^</code> and <code>\$</code> are able to match at the start or end of any line within the string.
both s and m modifiers	Treat string as a single long line but detect multiple lines. <code>.</code> matches any

Condition	Behavior
	character, even the <code>\n</code> . <code>^</code> and <code>\$</code> . However, they are able to match at the start or at the end of any line within the string.

Alternations

Sometimes, you need a regular expression to match different possible words or character strings. This is possible by using the alternation metacharacter (`|`). So, if you want to match any substring that contains the word `hi` or the word `hello`, then use the following expression:

```
'/hi|hello/'
```

Bear in mind that the expression tries the alternative choices from left to right trying to match the regular expression at the earliest possible point in the string. The following are some examples:

```
"black and white".contains('/black|gray|white/')
// matches black
"black and white".contains('/white|gray|black/')
// matches black. Even though white is the first
// alternative in the string, black matches
// earlier in the string.
"Bye!".contains('/b|by|bye|bye!/i')
// matches b
"Bye!".contains('/bye!|bye|by|b/i')
// matches bye!
```

The last example suggests that if some of the alternatives are prefixes of the others, they put the longest alternatives first. Otherwise, they will never match.

In some way, you can think of character classes as character alternations. So `'/[aeiou]/'` behaves like `'/a|e|i|o|u/'`.

Grouping

Parts of a regular expression can be grouped so that they are treated as a single unit. Parts of a regular expression are grouped by enclosing them with parenthesis (). Grouping a subexpression has many uses such as for alternation on part of the whole regular expression, for repetitions, for text extraction and for backreferencing. (Extraction and backreferencing are discussed in later lessons.)

Some grouping examples are displayed in the following table:

Regular Expression

```
'/(straw|blue|rasp)berry/'
```

String that Matches: strawberry, blueberry, or raspberry

Regular Expression

```
'/Blah( blah)*/'
```

String that Matches: Blah, Blah blah, Blah blah blah, Blah, blah, blah, blah,...

Regular Expression

```
'/^(a|b)/'
```

String that Matches: Matches either a or b at the beginning of the line (note that `'/^a|b/'` would match a at the beginning or any b

anywhere).

Regular Expression

```
`/y(es)?/i`
```

String that Matches: Y, y, or any case insensitive version of 'yes'.

Extraction

So far, you have been using regular expressions to see whether they match a string or part of a string. The grouping characters () serve another purpose: to extract part of the string that matches the regular expression. This is very useful for parsing and for general text processing.

For each grouping, the part of the string that matches inside the parenthesis goes into a particular position within the array of matched groupings. In FuegoBPM, the extraction can be done with the match() function, which returns the array of substrings for each grouping.

For example, suppose that you have a string with the current time, in hh:mm:ss format. You can build a basic regular expression for matching such as '/\d\d:\d\d:\d\d/'. However, you want to know what the values of the hour, minutes and seconds are. So, you group each of them with parenthesis. For example, '/(\d\d):(\d\d):(\d\d)/'. The following code shows how you display the hours, minutes and seconds using the index numbers of the arrays.

```
time as String
matches as String[]
input "Enter a time (hh:mm:ss):" time

matches = time.match('/(\d\d):(\d\d):(\d\d)/')

if matches is not null then
    display "Hours: " + matches[1] + "\n" +
           "Minutes: " + matches[2] + "\n" +
```



```
                "Seconds: " + matches[3]  
else  
    display "Invalid time!"  
end
```

Note



When a regular expression is matched against a string, the whole part of the string that matches is stored in position 0 (zero) of the array.

Remember that when a regular expression is matched against a string, the whole part of the string that matches is stored in position 0 (zero) of the array.

So, for the previous example, if you enter 12:40:23, the array will contain the following:

Position 0: 12:40:23

Position 1: 12

Position 2: 40

Position 3: 23

Positions are assigned to each group from left to right on the opening parenthesis.

Extraction – A Real World Example

The following is a real world example of extraction. Suppose that you need to interpret a text file with lines with the following format:

property = value

The file can also have comment lines, which begin with the pound sign (#). A sample of the file is as follows:

Configuration parameters

adminEmail=admin@yoursite.com

serverHost=server.yoursite.com

serverPort=12345

some preferences

soundEnabled=false

fontSize=12

colors

background = white

foreground = blue

What can you do? It would be useful if you could create an associative array, so that you can later do something like:

```
port = properties["serverPort"]
```

to get the value of the serverPort property defined in the file.

First, you need to define the regular expression to interpret a valid line in the file. As mentioned before, lines can be in property = value format or they may start with a # pound sign. In the latter case, the line must be ignored.

The assignment lines can be matched with `'/\w+=\w+/'`. This looks for a word (`\w+`) and equals sign (`=`) and another word (`\w+`).

Now, let's allow any optional white space around the equals sign:

```
' /\w+\s?=\s?\w+/'
```

That is right, but now you need to group the left side word (before

the equals) and the right side word (after the equals sign) so that you can extract the values.

```
' / (\w+) \s? = \s? (\w+) / '
```

One more detail is required. Let's force the regular expression to match the whole string. You achieve this by adding the `^` and `$` anchors.

```
' / ^ (\w+) \s? = \s? (\w+) $ / '
```

Let's test this one in the debugger.

```
input "Enter a line:" line
m = line.match('/^(\w+)\s?=\s?(\w+)$/')
if m is not null then
    display "Property: " + m[1] + "\nValue: "
        + m[2]
else
    display "ERROR, invalid line!"
end
```

Now, the comment lines should be allowed. A comment is easy to match by using the following regular expression (remember comment lines begin with the pound sign `#` in the sample text file):

```
' / ^#.* / '
```

`'/^#.*/'` means a line beginning with `#` and followed by any number of characters. Let's use an alternation to allow comment lines to match and test the Method again.

```
input "Enter a line:" line
m = line.match('/(^#.*$)|^(\\w+)\\s?=?\\s?(\\w+)$/')
if m is not null then
  if m[1] = "" then
    display "Property: " + m[3] + "\\nValue: "
      + m[4]
  else
    display "Comment line found: " + m[0]
  end
else
  display "ERROR, invalid line!"
end
```

Great! Now that you have tested the regular expression, you can remove the display statements and write the code that builds the associative array. Instead of reading the lines from an input, let's really read them from a file!

```
for each line in TextFile("/tmp/test.txt").lines
  m = line.match('/(^#.*$)|^(\\w+)\\s?=?\\s?(\\w+)$/')
  if m is not null then
    if m[1] = "" then           // if m is not a comment
      props[m[3]] = m[4]
    end
  else
    // erroneous line - ignore it
  end
end

display props
```

Obviously, replace "tmp/test.txt" with a valid file name and location before testing the Method with the Method Debugger.

Note



The TextFile component contains a built-in function for creating an associative array from a properties file. This example just shows you how to use regular expressions in a real problem. If the file were compatible with a Java properties file, then the Textfile.loadPropertiesFrom component

is the easiest solution.

The following table contains some more examples of text extraction:

Problem: To separate the path from the filename of a fully-qualified UNIX path and filename such as /usr/fuegotech4.0/readme.txt

Solution:

```
'/(.*)\$/([^\$]*)$/'
```

Result: [1] contains the path (/usr/fuegotech4.0 in the example [2] contains the name of the file (readme.txt in the example)

Problem:A simple way to extract the userID and the host name from an e-mail address such as sales@fuegotech.com

Solution:

```
'/([\w\.\ ]+)\@([\w\.\ ]+)/'
```

Result: [1] contains the userID (sales in the example) [2] contains the host name (fuegotech.com in the example)

Problem:To extract the different parts of a URL such as http://www.fuegotech.com:80/index.html.

Solution:

```
'/(\w+):\//([^\:\/]+)(:(\d+))?(\/.*)?/'
```

Result: [1] contains the protocol (http in the example) [2] contains the host name (www.fuegotech.com in the example) [3] contains the port number, including the colon (:80 in the example) [4] contains the port number (80 in the example) [5] contains the resource

(/index.html in the example).

Backreferencing

The substring matched by a grouping can also be referenced within the regular expression, which is known as backreferencing. Backreferencing allows you to make matches later in a regular expression depending on what matched earlier in the regular expression. You can reference a previous grouping with `\x`, where `x` is the grouping position.

The following table provides some backreferencing examples:

Regular Expression

```
'/(~np~\~/np~w+) ~np~\~/np~1/ '
```

String that Matches

The same word repeated twice. For example, the echo ha ha...

Regular Expression

```
'/(~np~\~/np~w+)~np~\~/np~1/ '
```

String that Matches

Words with repeated parts. For example, mama, papa, coco...

Regular Expression

```
'/(~np~\~/np~d)(~np~\~/np~d)(~np~\~/np~d)~np~\~/np~2~np~\~/np~1 '
```

String that Matches

Any five-digit palindrome number. For example, 12321, 83638, 91119...

Search and Replace

Regular expressions can also be used to replace parts of a string by a different string. In FuegoBPM Studio, you can do this using the `replace()` function. The first argument to `replace` is the regular expression search and the second argument is the new string. For example,

```
myString = "We played 1 on 1"
myString.replace('/1/', "one")

display myString
```

In the second line, 1 is replaced by one. Notice that only the first occurrence of 1 is replaced when you try the code with the debugger. In order to replace all occurrences, you must use the `g` modifier. For example,

```
myString.replace('/1/g', "one")
```

The following code strips any zero digit on the left end of a string. For example, if you want to change 005422 to 5422, you enter the following code.

```
myString.replace('/\b0+/g', "")
```

A powerful feature of the `replace` function is that you can use backreferencing in the replacing string. You do so by using the `$x`, where `x` is the grouping number.

For example, to swap the first two words in a string:

```
myString.replace('/(w+) (\w+)/', "$2 $1")
```

To convert a MM-DD-YYYY date to DD/MM/YYYY:

```
myString.replace('/(\d+)-(\d+)-(\d+)/', "$2/$1/$3")
```

To remove quotes from around any word:

```
myString.replace('/"(\w+)"/g', "$1")
```

Chapter 16. Code Conventions

Code Conventions

The present guide has an annotated form, thus making it far easier to be used during project code reviews than most other existing guidelines. In addition, programming recommendations generally tend to mix style issues with language technical issues in a confusing manner. This document does not contain any technical recommendations at all. Instead, it mainly focuses on programming style.

Layout of the Recommendations.

The recommendations are grouped by topic, the layout of the recommendations is as follows:

- Guideline short description
- Example, if applicable
- Motivation, background, and additional information

The motivation section is important. Coding standards and guidelines tend to start "religious wars" and it is important to state the background for the recommendation.

Recommendation Importance

In the guideline sections, the terms must, should, and can have a special meaning. A must requirement must be followed. A should requirement is a strong recommendation, and a can requirement is a general guideline.

- General Naming Conventions

- Specific Naming Conventions
- Statements
- Layout And Comments

General Naming Conventions

Names representing types or modules must be nouns and they must be written in mixed case starting with upper case.

```
Line, FilePrefix
```

Variable names must be in mixed case starting with lower case.

```
line, filePrefix
```

Makes variables easy to distinguish from types and effectively resolves potential naming collision as in the declaration *Line line*.

Names representing constants (or enum values) must be all uppercase using underscore to separate words.

```
MAX_ITERATIONS, RED, MONDAY
```

Names representing methods must be verbs and they must be written in mixed case starting with lower case.

```
find(), computeTotalWidth()
```

This is identical to variable names but, in FuegoBPM Studio, methods are already distinguishable from variables by their specific form.

Abbreviations and acronyms should not be uppercase when used as a name.

```
exportHtmlSource();      // NOT: exportHTMLSource();  
openDvdPlayer();         // NOT: openDVDPlayer();
```

Using all uppercase for the base name will trigger conflicts with the naming conventions given above. A variable of this type should be named `dvd`, `html`, and so on, which is obviously not very readable. Another problem is illustrated in the examples above. When the name is connected to another, the readability is seriously reduced. The word following the acronym does not stand out as it should.

Variables should not have prefixes or suffixes.

Given the fact that FuegoBPM Editor has already colored variables in a different way depending on their scope (local, instance, and so on), it is not necessary to add prefixes.

```
length as Int  
this.length = length
```

Generic variables should have the same name as their type.

```
assignTopic (topic : Topic)  
NOT assignTopic (value : Topic)  
NOT assignTopic (aTopic : Topic)
```

```
NOT assignTopic (x : Topic)
```

Reduces complexity by reducing the number of terms and names that are used. Also, this makes it easier to deduce the type given a variable name only.

If, for some reason this convention does not seem to fit, it is a strong indication that the type name is badly chosen.

Non-generic variables have a role. These variables can often be named by combining role and type:

```
startingPoint as Point  
centerPoint as Point  
loginName as Name
```

All names should be written in English.

```
fileName NOT nomArchivo
```

English is the preferred language for international development.

Variables with a large scope should have long names, and variables with a small scope can have short names.

Scratch variables used for temporary storage or indices are best kept short. A programmer reading such variables should be able to assume that its value is not used outside a few lines of code. Common scratch variables are:

integers	i, j, k, m, n, i1, i2
booleans	b,b1,b2

integers	i, j, k, m, n, i1, i2
reals	x,y,z,w
Strings	s,str

The name of the object is implicit and should be avoided in a method name.

```
line.length  
NOT line.lineLength
```

The latter seems natural in the class declaration but it proves superfluous in use, as shown in the example.

Specific Naming Conventions

The term "compute" can be used in methods in which something is computed.

```
computeAverage valueSet  
computeInverse matrix
```

Gives the reader the immediate clue that this is a potential time-consuming operation and, if used repeatedly, he/she might consider caching the result. Consistent use of the term enhances readability.

The term "find" can be used in methods where something is looked up.

```
findNearest Vertex  
findMinElementIn matrix  
NOT getMinElementIn matrix
```

Gives the reader the immediate clue that this is a simple look up method with a minimum of computations involved, **BUT** more expensive than a simple getter. Consistent use of the term enhances readability.

The term "initialize" can be used where an object or a concept is established.

```
initializeFontSetFor Printer
```

The American spelling of "initialize" should be used instead of the English "initialise". Abbreviation of "init" must be avoided.

"n" prefix should be used for variables representing a number of objects.

```
nPoints, nLines
```

The notation is taken from mathematics, where it is an established convention to indicate a number of objects.

If "number of" is the preferred statement, numberOf prefix can be used instead of just n. The *num* prefix must not be used.

"No" suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics, where it is an established convention for indicating an entity number.

Complement names must be used for complementary entities.

add/remove,

create/destroy,

start/stop,

insert/delete,

increment/decrement,

old/new,

begin/end,

first/last,

up/down,

min/max,

next/previous,

old/new,

open/close,

show/hide,

get/set

Reduce complexity by symmetry.

Abbreviations in names should be avoided.

```
computeSalary Employee  
NOT   compSal Employee
```

There are two types of words to be considered:

Common words listed in a language dictionary must never be abbreviated. Never write:

pt instead of point

comp instead of compute

init instead of initialize

etc.

There are also domain-specific phrases that are more naturally known through their acronym or abbreviation. These phrases should be kept abbreviated. Never write:

HypertextMarkupLanguage instead of html

CentralProcessingUnit instead of cpu

PriceEarningRatio instead of pe

etc.

Negated boolean variable names must be avoided.

```
isError as Bool    // NOT:    isNotError
isFound as Bool    // NOT:    isNotFound
```

The problem arises when the logical **not** operator is used and double negative arises. It is not immediately apparent what **not** `isNotError` means.

Exception classes should be suffixed with `Exception`.

```
AccessException
```


Exception classes are really not part of the main design of the code. Naming them like this makes them stand out relative to the other classes.

Functions (methods returning an object) should be named after what they return, and procedures (void methods), after what they do.

Increase readability. Makes it clear what the unit should do and, especially, what it is not supposed to do. Again, this makes it easier to keep the code clean of side effects.

Statements

Variables

Variables should be initialized where they are declared and they should be declared in the narrowest possible scope.

This ensures that variables are valid at any time.

Sometimes, it is impossible to initialize a variable to a valid value where it is declared. In these cases, it should be left uninitialized rather than initialized to some phony value.

Variables must never have dual meaning.

Enhance readability by ensuring that all concepts are uniquely represented. Reduce the chance of error by side effects.

Variables should be kept alive for as short a time as possible.

By keeping the operations on a variable within a narrow scope, it is easier to control the effects and side effects of the variable.

Loops

Loop variables should be initialized immediately before the

loop.

```
ready as Bool = true
while ready do
    //Do something
end
```

NOT:

```
ready as Bool = true

// Other stuff....

while ready do
    //Do something
end
```

The use of break/exit should be minimized.

These statements should be used only if they prove to give a higher readability than their structured counterparts.

Conditionals

Complex conditional expressions must be avoided. Introduce temporary boolean variables instead.

```
if (elementNo < 0) or (elementNo > maxElement)
    or elementNo = lastElement then
    //Do something
end
```

Should be replaced by:

```
isFinished as Bool = (elementNo < 0)
                    or (elementNo > maxElement)
isRepeatedElement as Bool = elementNo = lastElement
if isFinished or isRepeatedElement then
    //Do something
end
```

By assigning boolean variables to expressions, the program gets automatic documentation. The construction will be easier to read and debug.

The nominal case should be put in the if-part and the exception in the else-part of an if statement.

```
isError as Bool = readFile (fileName)

if not isError then
    //Do something
else
    //Handle the error
end
```

Makes sure that the exceptions do not obscure the normal path of execution. This is important for both the readability and performance.

Executable statements in conditionals must be avoided.

```
file = openFile (fileName, "w")
if file /= null then
    //Do something
end
```

NOT:

```
if (file = openFile (fileName, "w")) is not null then
    //Do something
end
```

Conditionals with executable statements are very difficult to read. This is especially true for new programmers.

Miscellaneous

The use of magic numbers in the code should be avoided.

If the number does not have an obvious meaning by itself, the readability is enhanced by introducing a named constant instead.

```
d = s / SECONDS_PER_DAY
timeout = DEFAULT_TIMEOUT
```

NOT:

```
d = s / 86400
timeout = 1000
```

Real and Decimal constants should always be written with a decimal point and at least one decimal.

```
total = 0.0
speed = 3.0

sum = (a + b) * 10.0;
```

NOT:

```
total = 0;  
speed = 3;  
  
sum = (a + b) * 10;
```

This emphasizes the different nature of integer and floating point numbers, even if their values might happen to be the same in a specific case.

Moreover, as in the last example above, it emphasizes the type of the assigned variable (sum) at a point in the code where this might not be evident.

Real and Decimal constants should always be written with a digit before the decimal point.

```
total as Real = 0.5f
```

NOT:

```
total as Real = .5f
```

The number and expression system in FuegoBPM Studio is borrowed from mathematics and one should adhere to mathematical conventions for syntax wherever possible. In addition, 0.5 is much more readable than .5. There is no way it can be mixed with the integer 5.

Layout And Comments

Basic indentation should be 4.

```
for i in 1..10 do
  a[i] = 0
end
```

The if-else class of statements should have the following form:

```
if condition then
  // statements
end
```

or:

```
if condition then
  statements
else
  statements;
end
```

or:

```
if condition then
  statements
elseif condition then
  statements;
else
  statements;
end
```

NOT:

```
if condition
```

```
then
  statements
end
```

NOT:

```
if condition then statements end
```

The chosen approach is considered to be better since each part of the if-else statement is written on separate lines of the file. This should make it easier to manipulate the statement, for instance, when moving else clauses around.

A bounded loop should have the following form:

```
for i in set do
  //statements
end
```

An unbounded loop should have the following form:

```
while condition do
  //statements
end
```

A Multipath conditional statement should have the following form:

```
case condition
when ABC
  // statements
```

```
when DEF
    // statements
else
    // statements
end
```

A Compound statement should have the following form:

```
do
    statements;
on e as Exception exception
    statements
end
```

or:

```
do
    statements;
on e as Exception
    statements
on exit
    statements
end
```

White Space

Spaces in expressions

- Conventional operators should be surrounded by a space character.
- Reserved words should be followed by a white space.
- Commas should be followed by a white space.

- Colons should be followed by a white space.
- Semicolons for statements should be followed by a space character.

```
a = (b + c) * d
```

Makes the individual components of the statements stand out and enhances readability. It is difficult to give a complete list of the suggested use of a white space in FuegoBPM Studio code. However, the examples shown above should give a general idea of the intentions.

Logical units within a compound statement should be separated by one blank line.

Enhances readability by introducing a white space between logical units of a block.

Comments

Tricky code should not be commented on but rewritten.

In general, the use of comments should be minimized by making the code self-documenting by appropriate name choices and an explicit logical structure.

All comments should be written in English.

In an international environment, English is the preferred language.

Avoid the use of multi-line comments.

```
// Comment spanning
```

```
// more than one line
```

Since nested multi-line comments are not supported, using single line comments ensures that it is always possible to comment out entire sections of code for debugging purposes, among others.

Comments should be indented in relation to their position in the code.

```
while true do
  // Do something
  something()
end
```

NOT:

```
while true do
// Do something
  something()
end
```

This is to avoid that the comments break the logical structure of the program.

Files

File content must be kept within 100 columns.

The incompleteness of split lines must be made obvious.

```
totalSum = a + b + c +
           d + e
```

```
function (param1, param2,  
        param3)  
  
passingText ("Long line split" +  
            "into two parts.")
```

Split lines occur when a statement exceeds the column limit given above. It is difficult to provide strict rules for the manner how lines should be split, but the examples above should give a general hint.

In general:

- Break after a comma.
- Break after an operator.
- Align the new line with the beginning of the expression on the previous line.

Chapter 17. Refactories

Refactories

Nested conditional statements

Nested conditional statements are automatically grouped in one statement with both conditions.

Before

```
a as Int
if a > 2 then
  if a < 10 then
    a = 5
  end
end
```

After

```
a as Int
if a > 2 and a < 10 then
  a = 5
end
```

Identifiers for Exceptions

Identifiers for exception handlers are automatically added when they are not specified.

Before

```
message as String
do
  message = "Ok"
on Exception
  message = Exception.message
end
```

After

```
message as String
do
  message = "Ok"
on e as Exception
  message = e.message
end
```

Bounded loop instead of unbounded loop

Unbounded loops are converted to bounded loops when possible.

Before

```
i = 0
while i <= 10 do
  i = i + 1
end
```

After

```
for i in 0..10 do
end
```

Conditional Exit

Conditional statements with an exit statement are transformed to conditional exits.

Before

```
array as Int[] = [10, 20, 30]

for each e in array do
  if e = 20 then
    exit
  end
end
```

After

```
array as Int[] = [10, 20, 30]

for each e in array do
  exit when e = 20
end
```

Redundant negation

Redundant negations are removed.

Before

```
a as Int = 5

if not a != 2 then
end
```

After

```
a as Int = 5  
  
if a = 2 then  
end
```

Conditional statement inside else blocks

Before

```
a as Int = 2  
if a < 2 then  
    a = 2  
else  
    if a > 5 then  
        a = 5  
    end  
end
```

After

```
a as Int = 2  
  
if a < 2 then  
    a = 2  
elseif a > 5 then  
    a = 5  
end
```

Check for null value

Before

```
s as String
```

```
if s != null then  
end
```

After

```
s as String  
  
if s is not null then  
end
```

Right order for 'is not'

Before

```
s as String  
  
if not s is null then  
end
```

After

```
s as String  
  
if s is not null then  
end
```

Redundant equality

Before


```
found as Bool  
  
if found = false then  
end
```

After

```
found as Bool  
  
if not found then  
end
```

Explicit argument names

Before

```
open TextFile using "", ""
```

After

```
open TextFile using name = "",  
                    lineSeparator = ""
```

Unneeded parenthesis

This refactory is applied to 'if' and 'while' conditions in Fuego Business Language (FBL).

Before

```
a as Int = 2  
if (a > 2) then  
end
```

After

```
a as Int = 2  
if a > 2 then  
end
```

Legacy multi path conditional statements

Before

```
a as Int = 2  
switch a in  
  case 2:  
    display "Two"  
  case 4:  
    display "Four"  
end
```

After

```
a as Int = 2  
case a  
when 2 then  
  display "Two"  
when 4 then  
  display "Four"  
end
```

Functions

Functions are rewritten using functional syntax.

Before

```
a as Int  
a = a.abs()
```

After

```
a as Int  
a = abs(a)
```

Wrong symbols

In FBL, some invalid symbols (e.g: &&, ||, !, ==) are accepted but they are automatically fixed when the code is rewritten.

Before

```
a as Int = 2  
b as Int = 4  
  
if ((a > 2 && a < 10) || b == 4) then  
end
```

After

```
a as Int = 2
b as Int = 4

if (a > 2 and a < 10) or b = 4 then
end
```

Misspelled member names

Before

```
Open TextFile using name = "",
                  lineSeparator = ""

Mail.content_type = ""
```

After

```
open TextFile using name = "",
                lineSeparator = ""

Mail.contentType = ""
```

Methods equals() and toString()

Before

```
if object.equals(this) then
    string = object.toString()
end
```

After

```
if object = this then  
  string = String(object)  
end
```