

# GNU Enterprise

---

Application Developer's Guide  
Edition 0.5.2, 2005-10-11

Reinhard Müller

---

Copyright © 2004 Free Software Foundation

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

# Table of Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>General Concepts Used in GNU Enterprise</b>	<b>2</b>
2.1	Classes	2
2.2	Modules	2
2.3	Class Extension	2
<b>3</b>	<b>Basic Database Features</b>	<b>4</b>
3.1	General GCD Layout	4
3.2	Using GCD Files	4
3.3	Modules	5
3.4	Classes	5
3.5	Properties	6
3.6	Data Types	6
3.7	Implicit properties	7
<b>4</b>	<b>Basic User Interface Features</b>	<b>8</b>
4.1	General GLD Layout	8
4.2	Using GLD Files	8
4.3	Multiple Languages	9
<b>5</b>	<b>Referencing Other Classes</b>	<b>11</b>
5.1	Defining Several Classes In A Module	11
5.2	Reference Properties	11
5.3	How References Are Displayed In Forms	12
<b>6</b>	<b>Defining And Using Procedures</b>	<b>13</b>
6.1	Triggers	13
6.1.1	OnInit	13
6.1.2	OnChange	13
6.1.3	OnValidate	14
6.1.4	OnDelete	14
6.2	Calculated Properties	14
6.3	Other Procedures	15
6.3.1	Defining Parameters And The Return Value	15
6.3.2	Calling Procedures From Other Procedures	16
6.3.3	Calling Procedures From Forms	16
6.4	Global Functions And Variables Available In Procedures	16
6.4.1	self	16
6.4.2	session	17
6.4.3	abort	17
6.5	Complex Queries Using Condition Trees	18

# 1 Introduction

This chapter still has to be written.

## 2 General Concepts Used in GNU Enterprise

### 2.1 Classes

In the past, business applications have often been designed in a classical client/server architecture, which means that a database server was used for pure data storage, while all business logic was put in the front end. Consequently, for applications accessing the same data through different ways (several forms and reports pointing to the same table), this meant implementing the same logic over and over again.

As a result, people started to move parts of the logic into the database server, using sophisticated SQL databases with triggers and stored procedures. However, this required writing the logic in a programming language specific to the database server, usually based on SQL, which was hard to learn and very limited in its possibilities.

With GNU Enterprise, we use an application server to handle both the data storage and the logic. While it delegates most of the data storage tasks to a classic SQL database backend, it lets you define the underlying logic by writing program code in Python, a very powerful yet easy to learn language.

In GNU Enterprise, a database table and the related program code form a logical unit called a *class*. A typical example for a class would be 'customer', being composed of a database table with columns like 'name', 'street', and 'city', and several pieces of program code defining things like under which conditions the customer may receive goods, how an address label has to be printed, what makes up a correct and complete customer record at all, or under which conditions the customer may be deleted from the database.

To distinguish between the database layer and the GNU Enterprise layer, we speak of *tables*, and *columns* for the database, and of *classes*, *properties*, and *procedures* for the related information in GNU Enterprise.

If you have ever dealt with object oriented programming, the word *class* probably rings a bell for you. But please be aware: although classes in GNU Enterprise share some aspects of object oriented programming (like the association of code with data or some kind of polymorphism), there are still some important differences. Most notably, GNU Enterprise classes don't support inheritance.

### 2.2 Modules

One of the main design goals of GNU Enterprise was to be *modular*. Several authors should be able to develop applications independently of each other, and these applications should be able to work together smoothly. To reach this goal, it was necessary to introduce a concept of *namespaces* so that class names can't collide.

Usually, a set of classes that is usually used together is put into the same namespace, maintained by the same author (or group of authors) and installed together. Such a set of classes form a *module*.

Typical examples of modules could be "general ledger" (consisting of an account class and some transaction classes) or "invoicing" (consisting of customer, item, invoice head and invoice item). Of course, modules could also build upon each other, so instead we could also have a "stock" module defining (possibly among others) an item class, while the "invoicing" module would reuse that item class and only add the other necessary classes.

### 2.3 Class Extension

As mentioned above, GNU Enterprise classes doesn't support the concept of inheritance. However, often a new module might want to change the behaviour of an existing class.

Given our above example, the "invoicing" module would not want to simply use the "item" class of the "stock" module, but it would want to add a rule to that class to make sure that, say, an item can't be deleted when it has been used in an invoice.

Using *class extension*, a module can not only reuse a class of another module, it can also change the behaviour of that class by adding new properties and new procedures to that class. Several modules can extend the same base class, and all changes are in effect no matter from where the class is accessed.

So if, for example, a "purchasing" module would add even another rule to the "item" class, namely that no item may be deleted if it was used in a purchase order, both tests (the invoice test and the purchase order test) will be in effect as soon as these additional modules are installed, no matter *who* tries to delete an item – even if the code that wants to delete an item was written by somebody not even aware that the invoicing and purchasing modules exist at all.

## 3 Basic Database Features

Let's start with a simple address management application.

For now, we will assume that `gnue-appserver` has been installed properly and is already running, that `gnue-forms` and `gnue-reports` are also installed, and that your `connections.conf` file is set up correctly to allow connections to the running application server with the `[appserver]` connection.

### 3.1 General GCD Layout

To define the classes for GNU Enterprise, we use *GCD files* (GNUe Class Definition files). These files are a specialized XML file format. You can write GCD files with any editor you want.

So for a start, let's look at a very simple GCD file:

```
<module name="address">
  <class name="person">
    <property name="name" type="string(35)" />
    <property name="street" type="string(35)" />
    <property name="zip" type="string(8)" />
    <property name="city" type="string(35)" />
  </class>
</module>
```

### 3.2 Using GCD Files

After having written the above GCD file, save it as `'address.gcd'` and stop `gnue-appserver`. Now we have to make GNU Enterprise read and process our class definition. We do this by typing

```
gnue-readgcd address.gcd
```

at the command prompt. This will teach GNU Enterprise about our new class and make the database backend create the tables necessary to hold the data of this class. You can re-run `gnue-readgcd` as often as necessary and the database will be inserted/updated, but elements will not be deleted. Restart `gnue-appserver`.

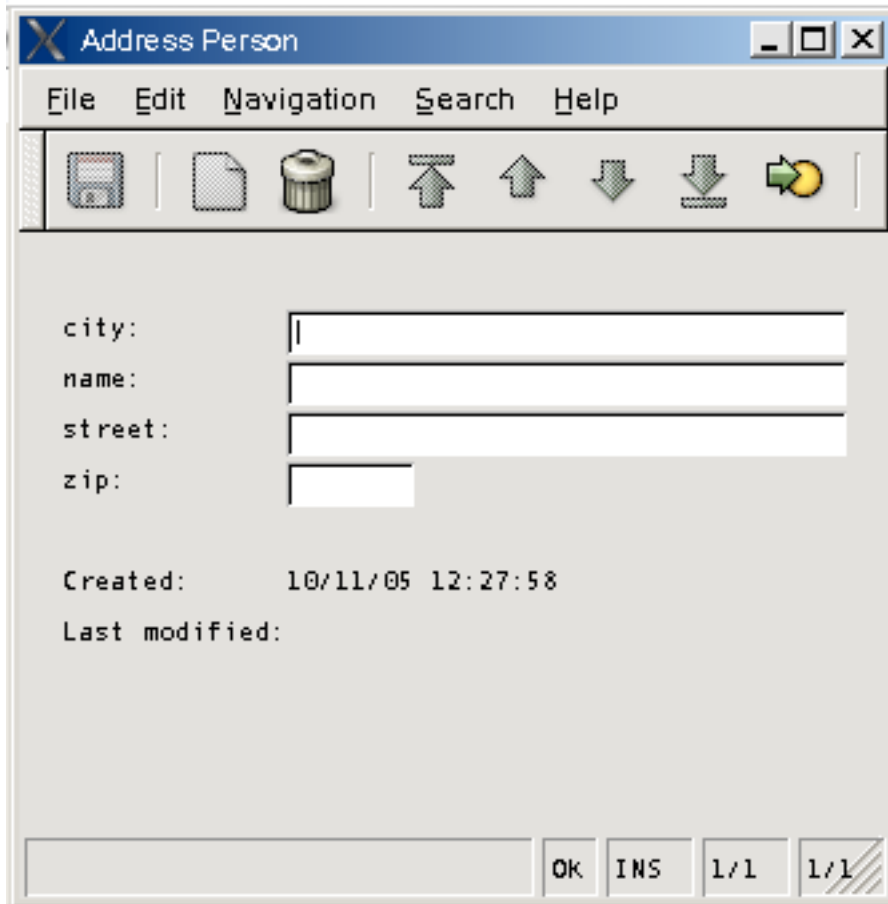
Please note that some databases may allow adding new tables while there is an active connection to the database server. You may not have to stop `gnue-appserver` depending on the database selected.

As an alternative, GNU Enterprise supports a configuration option called the *module path*. You can simply place your GCD files (and also the GLD files we will learn about later) into any directory given in that path or into a subdirectory of any of these directories, send the `gnue-appserver` process a `SIGHUP` signal, and GNU Enterprise will read in the GCD files automatically. However, this does not work under Windows.

Now, as GNU Enterprise has learned about our class, we are ready to display a form and enter data: type

```
gnue-forms appserver://appserver/form/address_person
```

and enjoy your first application!



Please note the syntax of the last part in the command line, consisting of the module name, an underscore, and the class name. We call this the *fully qualified class name*.

You will probably notice that the form doesn't look very good yet. Of course we will improve that later, but first let us analyze the content of our little example file so far.

### 3.3 Modules

The outermost level in the XML hierarchy, the *root element*, is the **module**. Everything we define in a GCD file is defined within a module, and you can have exactly one module per GCD file.

The following attributes are valid for the **module** tag:

- |                |  |
|----------------|--|
| <b>name</b>    | unique name of the module, namespace identifier for everything defined within the module (max. 35 characters, but usually not more than 10 characters, can not include underscore character, required) |
| <b>comment</b> | explanatory text, not used by the system in any way (max. 70 characters, optional)   |

### 3.4 Classes

Naturally, we start the module by defining a **class**. The class tag can have the following attributes:

<b>name</b>	identifier for this class, has to be unique within the module (the length of this name plus the length of the module name should not exceed 30 characters, can not include underscore character, required)
<b>comment</b>	explanatory text, not used by the system in any way (max. 70 characters, optional)
<b>module</b>	may only be given if a class of a foreign module is extended; don't let this confuse you, this will be explained later, we've just put it here for the sake of completeness

### 3.5 Properties

The most important building blocks of a class definition are **property** tags, defining the data fields of the class. These attributes can be given for a **property** tag:

<b>name</b>	identifier for the property, has to be unique within the class (except for the case of class extension) (the length of this name plus the length of the module name should not exceed 30 characters, required)
<b>comment</b>	explanatory text, not used by the system in any way (max. 70 characters, optional)
<b>type</b>	data type of this property, see below (required)
<b>length</b>	maximum length of data stored in this property, if appropriate for the data type (optional)
<b>scale</b>	scale of data stored in this property, if appropriate for the data type (optional)
<b>nullable</b>	'true' means the value of <b>None</b> is valid for this property, while 'false' means it is not (optional, defaults to 'true')

### 3.6 Data Types

GNU Enterprise supports the following basic data types:

<b>string</b>	Strings can hold arbitrary text. A length may be given and means a maximum length of the text; if no length is given, maximum length is limited only by the database backend in use. As long as the backend supports Unicode, there is no limitation on the characters that may appear in the string.
<b>number</b>	Numbers can be defined with length only (meaning integers with the given maximum number of digits) or with length and scale (meaning a maximum number of digits as given in length, where exactly the number of digits given in scale are after the decimal point).
<b>boolean</b>	Booleans can only hold <b>TRUE</b> or <b>FALSE</b> . Neither length nor scale may be given for booleans.
<b>date</b>	The valid range of dates depends on the capability of the database backend. Neither length nor scale may be given for date properties.
<b>time</b>	This stores an absolute time value and is heavily dependent on the database backend. Future versions of GNU Enterprise might have features to unify the handling of time values for the different databases, but until then, time properties are not recommended. Neither length nor scale may be given for time properties.

**datetime** Datetime properties store a date and a time, often also called a *timestamp*. The valid range as well as the resolution depends on the database backend, and it is generally not recommended to use fractions of seconds as not all databases support this. Neither length nor scale may be given for datetime properties.

Length and scale may also be given in the `type` attribute, after the name of the type, enclosed in parentheses, with a comma between length and scale. So

```
<property name="foo" type="string(35)" />
<property name="bar" type="number(12,2)" />
```

is equivalent to

```
<property name="foo" type="string" length="35" />
<property name="bar" type="number" length="12" scale="2" />
```

### 3.7 Implicit properties

For every class, GNU Enterprise implicitly creates a few properties:

**gnue\_id** is a non-changeable string property with 32 characters. GNU Enterprise automatically assigns a unique random string for every new instance that is created. The value of `gnue_id` will remain constant for a given instance until it is deleted. The underlying column in the database table, also called `gnue_id`, serves as the primary key for the table.

**gnue\_createdate**  
holds the date and time of the creation of an instance.

**gnue\_createuser**  
holds the user name that created the instance, in case GNU Enterprise is configured for user authentication.

**gnue\_modifydate**  
holds the date and time of the last modification to this instance. It can be empty if an instance has just been created and is still in virgin state.

**gnue\_modifyuser**  
holds the user name that did the last modification to this instance, in case GNU Enterprise is configured for user authentication. It can be empty if an instance has just been created and is still in virgin state.

## 4 Basic User Interface Features

As we are now able to define our classes with all kinds of data to be held in them, as a next step we want to create a better interface for the user to enter the data.

### 4.1 General GLD Layout

GNU Enterprise can use a lot of information from our class definition to build a form usable for entering data into that class. However, some decisions cannot be based on the class definition, most notably a good decision about field order and about labels to display.

For that purpose, we have a second file type in GNU Enterprise: *GLD files* (GNUe language definition files). Like GCD files, they are a specialized form of XML.

Let's create a GLD file matching our example:

```
<module name="address" language="C">
  <class name="person" label="Address management">
    <property name="name" pos="100" label="Name" />
    <property name="street" pos="200" label="Street" />
    <property name="zip" pos="300" label="Zip-Code" />
    <property name="city" pos="400" label="City" />
  </class>
</module>
```

### 4.2 Using GLD Files

After having written the above GLD file, save it as 'address-C.gld'. Like for GCD files, there's a command to read GLD files into GNU Enterprise:

```
gnue-readgld address-C.gld
```

(please note the difference: *gnue-readgcd* for GCD files and *gnue-readgld* for GLD files)

Alternatively, on Posix compatible systems, you can place your GLD files into the module path defined in GNU Enterprise's configuration, and send the *gnue-appserver* process a SIGHUP signal.

Now we can see how GNU Enterprise displays a much nicer form:

```
gnue-forms appserver://appserver/form/address_person
```

The `label` tag of the class defines the form title, the `label` tag of the properties tells GNU Enterprise about the labels to use for the several fields in the form. Fields are ordered by the number given in the `pos` attribute of the properties. Properties with `pos="0"` are not shown in the form at all.

### 4.3 Multiple Languages

GNU Enterprise was designed from the beginning to support multiple languages. If English isn't your native language, and you want GNU Enterprise to display the form with translated labels. No problem: just create a second file, for example `'address-de.gld'` for German texts, exchange all labels with their respective translation, and read that into GNU Enterprise, too.

For example:

```
<module name="address" language="de">
  <class name="person" label="Adressverwaltung">
    <property name="name" pos="100" label="Name" />
    <property name="street" pos="200" label="Straße" />
    <property name="zip" pos="300" label="Postleitzahl" />
    <property name="city" pos="400" label="Stadt" />
  </class>
</module>
```

Now, users will be presented with the interface according to their own locale setting.

The "-de" in the file name is a convenience to recognize language files. GNU Enterprise uses the attribute "language" in the "module" tag to define the language.

## 5 Referencing Other Classes

So far, we have only defined a single class in our module. However, a database design in practice usually consists of several classes, and these classes reference each other. We will now see how we define several classes in a module and their relationship with each other.

### 5.1 Defining Several Classes In A Module

Definition of more than one class within a module is very straightforward - just include a `<class>` tag for each class within a single `<module>` tag in the GCD file:

```
<module name="address">
  <class name="country">
    <property name="code"    type="string(2)" comment="ISO-Code" />
    <property name="name"    type="string(35)" />
  </class>
  <class name="person">
    <property name="name"    type="string(35)" />
    <property name="street"  type="string(35)" />
    <property name="zip"     type="string(8)" />
    <property name="city"   type="string(35)" />
  </class>
</module>
```

The same works for GLD files:

```
<module name="address" language="C">
  <class name="country" label="Country Codes">
    <property name="code"    pos="100" label="ISO Code" />
    <property name="name"    pos="200" label="Name" />
  </class>
  <class name="person" label="Address management">
    <property name="name"    pos="100" label="Name" />
    <property name="street"  pos="200" label="Street" />
    <property name="zip"     pos="300" label="Zip-Code" />
    <property name="city"   pos="400" label="City" />
  </class>
</module>
```

Please note this will result in two separate forms you will be able to call:

```
gnue-forms appserver://appserver/form/address_country
gnue-forms appserver://appserver/form/address_person
```

Don't forget that you have to run `gnue-gcd` after you have changed a GCD file, and `gnue-gld` after you have changed a GLD file, or you have to put the files into GNU Enterprise's module path and send the `gnue-appserver` process a `SIGHUP` signal!

### 5.2 Reference Properties

Of course we will now want to link our two classes together: The person class should not only store the street, the zip and the city, but also the country to make up a complete address.

We achieve this in adding a new property to the person class. But this new property will not contain a string with the country code or name, it will rather just *reference* an instance of the country class.

A reference property is defined in the GCD file by giving the referenced class as type of the property:

```
<property name="country" type="address_country" />
```

And again we see that *fully qualified classname*, consisting of the module name, an underscore, and the class name.

In the GLD file, a reference property is handled similar to a normal property:

```
<property name="country" pos="500" label="Country" />
```

However, there's something missing: What should the form display for the country? Should the country code be displayed? The name of the country? Both of them? How will the user see what countries are valid?

### 5.3 How References Are Displayed In Forms

The most usual concept here is to display a dropdown in the form that lists all available countries, and to let the user select one of them. But still we have to define which property of the 'country' class should be listed in the dropdown. This is done in the GLD file for the 'country' class:

```
<class name="country">
  <property name="code" pos="100" label="ISO Code" />
  <property name="name" pos="200" search="100" label="Name" />
</class>
```

The 'search="100"' defines that exactly this property will be listed in the dropdown for the user to select from. You can try to move that part to the 'code' property, and you will see that the dropdown will be filled with country codes instead of country names.

Even more, you can have more than one property in a class with a **search** attribute: this will result in several dropdowns being displayed side by side, in the order of the number given for the **search** attribute. Try making the code property 'search="100"' and the name property 'search="200"' and look at the result. Then, try the other way around. You will see how flexible this system is.

If you want only the code to appear as a dropdown and the name to be just shown as a non-editable text field next to it, try 'search="100"' for the code and 'info="200"' for the name: **info** is handled just like **search**, except that it doesn't show as a dropdown, but as a simple informative text.

Please note that all this is defined in the GLD for the *referenced* class, in our case the 'country' class. That means that *every other class* referencing a country will show the *same* dropdowns in the user interface.

Additionally, a form generated by GNU Enterprise will always order the records by the property that was given the lowest number in the **search** attribute.

## 6 Defining And Using Procedures

As mentioned before, procedures are the second element classes in GNU Enterprise are built upon. GNU Enterprise is designed to generally support different programming languages for procedures, and while Python is the only language currently supported, others might follow.

Procedures can have zero, one, or more than one parameters, and they may have or have not a return value. All parameters and the return value must be of a fixed datatype.

Procedures are defined in the GCD file.

### 6.1 Triggers

*Triggers* are procedures that are called automatically by the GNU Enterprise system on occurrence of predefined events. Triggers have no parameters and no return value.

#### 6.1.1 OnInit

If a class contains a procedure with the name `OnInit`, the code in this procedure is automatically executed whenever a new instance of this class is created.

`OnInit` procedures are usually used to initialize properties with default values. An instance that has seen no changes except that `OnInit` has run is still considered empty and is not affected by a commit. In other words: Changes to an instance done in `OnInit` will be silently discarded when this instance is not saved afterwards. `OnInit` *may not have side effects!*

This example initializes the name for newly created person instances with "Foo".

```
<module name="address">
  <class name="person">
    <property name="name"    type="string(35)" />
    <property name="street"  type="string(35)" />
    <property name="zip"     type="string(8)" />
    <property name="city"    type="string(35)" />

    <procedure name="OnInit" >
      self.name = 'Foo'
    </procedure>
  </class>
</module>
```

#### 6.1.2 OnChange

If a class contains a procedure with the name `OnChange`, the code in this procedure is executed whenever a property is changed.

The code in this procedure can access the variable `oldValue` to access the value before the change, `newValue` to access the value that the property should be changed to, and `propertyName` to see what property is being changed.

`OnChange` is executed once for every single property that is changed. If several properties are changed in a single step, `OnChange` is executed for every property sequentially.

All assignments to properties done in `OnInit` are considered setting default values, not changing something. `OnChange` is not executed for all these assignments. In other words, `OnChange` is never fired from `OnInit`.

It is possible to call the function `abort` in `OnChange`. This function exits the current procedure and prevents the change to the property to happen. `abort` takes a single parameter, the error message to be presented to the user.

The following example shows how to make sure that no zip codes greater than 10000 can be entered:

```
<module name="address">
  <class name="person">
    ...

    <procedure name="OnChange" >
      <![CDATA[
        if propertyName == 'address_zip':
          if newValue and int (newValue) < 10000:
            abort ('ZIP codes must be less than 10000')
      ]]>
    </procedure>
  </class>
</module>
```

Everything inside a CDATA section is ignored by the parser.

If your text contains a lot of "<" or "&" characters - as program code often does - the XML element can be defined as a CDATA section.

A CDATA section starts with <![CDATA[ and ends with ]]>

### 6.1.3 OnValidate

If a class defines a procedure with the name `OnValidate`, the code in this procedure is executed right before a commit.

Every commit triggers the execution of `OnValidate` for all affected instances in more or less random order. Changes done in `OnValidate` will become persistent with the current commit. However, if the code in `OnValidate` affects other instances, these changes are not guaranteed to run through the `OnValidate` of the other class (but this might change in the future).

`OnValidate` can prevent the commit to happen if it raises an exception with `abort` or by other means.

Typical uses of `OnValidate` are calculation of redundant properties from other properties, automatic filling of properties that has side effects and therefore may not happen in `OnInit` (like getting the next sequential number), and of course validation.

### 6.1.4 OnDelete

If a class contains a procedure named `onDelete`, the code in this procedure is executed whenever an instance is about to be deleted.

`onDelete` can prevent the deletion to happen if it raises an exception with `abort` or by other means.

Typical uses of `onDelete` are to prevent deletion of instance under specific conditions, or to automatically delete detail records along with a master.

## 6.2 Calculated Properties

Apart from the properties that are stored in the database, GNU Enterprise supports a kind of virtual properties that are determined at runtime as the result of a procedure. We call them *calculated properties*.

The following example shows a very simple calculated property that is built as a concatenation of country code, zip, and city.

```

<module name="address">
  <class name="country">
    <property name="code"      type="string(2)" comment="ISO-Code" />
    <property name="name"      type="string(35)" />
  </class>
  <class name="person">
    <property name="name"      type="string(35)" />
    <property name="street"    type="string(35)" />
    <property name="country"   type="address_country" />
    <property name="zip"       type="string(8)" />
    <property name="city"      type="string(35)" />

    <property name="czc"       type="string(47)" >
      return self.country.code + ' ' + self.zip + ' ' + self.city
    </property>
  </class>
</module>

```

The possibilities of calculated properties are virtually unlimited. You can access all other properties of the instance (even other calculated properties), properties of related or unrelated other instances, and you can even call external programs. It is strongly advised, however, that the code in calculated properties does not have any side effects.

The type of calculated properties (which is effectively the type of the return value of the assigned procedure code) can be any of the basic data types (string, number, boolean, date, time, and datetime). It is not possible to pass object references as the result of calculated properties.

Whenever a calculated property with the name 'foo' is defined, GNU Enterprise defines a procedure with the name 'getfoo', and every access to the property 'foo' is automatically and transparently translated into a call of the 'getfoo' procedure. However, the only reason why you need to know this at all is that you should not start the name of any of your procedures with `get`.

The labels for calculated properties are defined in a GLD file just like for other properties.

## 6.3 Other Procedures

Apart from triggers and calculated properties, arbitrary procedures can be defined with names not starting with `on` or `get` (independent from case). These procedures may or may not have parameters, and they may or may not have return values, and they may contain virtually anything.

The return value of other procedures (the type of the procedure) can be any of the basic data types (string, number, boolean, date, time, and datetime). It is not possible to pass object references as the result of procedures.

### 6.3.1 Defining Parameters And The Return Value

Parameters are declared in the `gcd` file by adding one or more `<parameter>` tags within the `<procedure>` tag, with `name`, `type`, `length`, `scale`, and `comment` attributes just like you would use them for properties.

The `type`, `length`, and `scale` attributes are also allowed directly in the `<procedure>` tag, where they declare the return type of the procedure.

### 6.3.2 Calling Procedures From Other Procedures

Within procedure code (also within the code of triggers and calculated fields), all defined procedures are available as methods of the respective objects.

In Python, all parameters must be given as keyword arguments.

The following examples show a few ways of calling procedures:

```
self.foo ()
self.bar (baz = 17)
self.customer.doit (really = True)
self.invoice.customer.doSomething (text = "foo", other = "bar")
```

In the last line of the example above we see that:

`self.invoice` would be an object, `customer` a property of that object pointing to a third object, and that third object has a procedure called `doSomething`.

### 6.3.3 Calling Procedures From Forms

If you you author your own form definition (.gfd) file, you can call a procedure of a class if that class in form trigger code. You must have a block assigned to that class.

To call a procedure, call the function `call` of the block and pass two parameters: the first one is the (fully qualified) procedure name to call, and the second one is a dictionary with `{'parametername': 'value'}` pairs.

## 6.4 Global Functions And Variables Available In Procedures

In the code of any procedure, you can use these variables and global functions:

### 6.4.1 self

The variable `self` allows access to the object instance the procedure is called for. `self` is usable just like any normal Python object: you can access (read and write) properties of the object as `'self.propertyname'` and you can call procedures like object methods as `'self.methodname (parameters)'`.

Reference properties virtually hold the objects themselves, so you can do things like

```
c = self.customer
self.custname = c.name
c.lastinvoice = self
```

(where `self.customer` and `c.lastinvoice` would be reference properties)

To clear the value of a property (set it to NULL in database speak), you can assign it the value `None`. Accessing a property that is not set (or has been cleared this way) also yields `None` in case it's of one of the basic types.

However, unset reference properties yield a special object called `NoneObject` that simply returns `None` on access to any property. This way, you accessing for example `self.customer.name` for an object where the customer property is not set returns `None` instead of generating an exception.

Because of this, you may not test reference properties like `if self.customer is None:`, use `if self.customer:` instead.

Apart from all properties and procedures defined in the class definition or created implicitly by GNU Enterprise (like `gnue_id`), you can call the `delete ()` method of an object to delete it.

### 6.4.2 session

The variable `session` holds an object representing the current connection to the GNU Enterprise Application Server. It has a few handy methods you can use in your procedures:

`new (classname)` [Method on `session`]  
Creates and returns a new instance of the class `classname`.

`get (classname, objectId)` [Method on `session`]  
Returns the existing instance of the class `classname` that has a `gnue_id` of `objectId`. If no such instance exists, an exception is raised.

`find (classname, conditions = conditions, sortorder = sortorder, properties = properties)` [Method on `session`]  
Returns a list of those instances of class `classname` that match the `conditions`, ordered by `sortorder`.

The conditions can be given as a dictionary of propertyname/value pairs (in which case all the properties must have exactly the given value to match) or as a complex condition in prefix notation. The default is to return all instances of the given class.

The sort order is a list of sort items. An item can be either a simple string (interpreted as a property name) or a dictionary with the keys 'name' (required, the property name), 'descending' (optional, `True` indicates descending sort order), and 'ignorecase' (optional, `True` indicates the sort should be case insensitive). The default is to sort by the `gnue_id` property (which is pretty useless in most cases).

The last optional parameter is a list of property names that should be prefetched from the database. Of course *any* property of the objects can be accessed at *any* time, but properties given in this list are included in the original database query and cached by GNU Enterprise, so subsequent access to prefetched properties is much faster.

The result of this function is an object that behaves very much like it is a list of the found objects, most notably it supports the following operations:

- testing the truth value to find out whether the list contains any items at all
- requesting the length of the list with the `len (list)` function (However, be aware that this can be a very time consuming process, especially if the condition contained calculated fields or `exists` operations)
- accessing the *n*th item in the list with `list [n]`
- accessing the *n*th to last item in the list with `list [-n]` (as this involves finding out the length of the list, it can also be a time consuming process under some circumstances)
- iterating through the list with a `for` loop

The returned list is immutable. The items of the list can be changed, but it is not possible to add items to the list or remove items from the list.

Changing any item of the list in a way that it would not any more match the condition originally given to `find ()` does not remove the item from the list. Likewise, changing an object that is not a member of the list in a way that it now would match the condition doesn't add it to the list, neither does creating a new instance. You have to call `find ()` again to get a current list of matching objects.

### 6.4.3 abort

This function aborts the current operation. It takes a single parameter, which is a string holding an error message to display to the user.

## 6.5 Complex Queries Using Condition Trees

The `find` function introduced above can do queries of all kinds using a powerful and flexible construction called the *condition tree*.

Condition trees are generally constructed as a sequence where the first element is an operation and the following elements are the arguments. For many operations, one or several of the arguments can again be an operation, thus building up a tree of operations.

The following operations are possible:

'Field'	Takes a single argument with a property name and evaluates to the value of this property.
'Const'	Takes a single argument with a constant value and evaluates to this value given in the argument.
'and'	Takes an arbitrary number of arguments of boolean type and evaluates to <code>True</code> if all arguments are <code>True</code> and to <code>False</code> otherwise.
'or'	Takes an arbitrary number of arguments of boolean type and evaluates to <code>True</code> if at least one of the arguments is <code>True</code> and to <code>False</code> otherwise.
'not'	Takes exactly one argument of boolean type and evaluates to <code>True</code> if the argument is <code>False</code> and vice versa.
'add'	Takes an arbitrary number of arguments of numeric type and evaluates to the sum of the arguments.
'sub'	Takes an arbitrary number of arguments of numeric type and evaluates to the first argument minus the sum of the rest of the arguments.
'mul'	Takes an arbitrary number of arguments of numeric type and evaluates to the product of the rest of the arguments.
'div'	Takes an arbitrary number of arguments of numeric type and evaluates to the first argument divided by the product of the rest of the arguments.
'negate'	Takes a single argument of numeric type and evaluates to the negation of the argument.
'eq'	Takes exactly two arguments of any type and evaluates to <code>True</code> if both arguments are equal and to <code>False</code> otherwise. If the arguments are of different types, they are tried to convert to the same type, where strings are converted to booleans, numbers, or date/time values, and booleans are converted to numbers. If such a conversion fails, an exception is raised.
'ne'	Takes exactly two arguments of any type and evaluates to <code>False</code> if both arguments are equal and to <code>True</code> otherwise. The same conversions as for <code>'eq'</code> apply.
'gt'	Takes exactly two arguments of any type and evaluates to <code>True</code> if the first argument is greater than the second one. If the arguments are strings, "greater" means "sorting after". If the arguments are date/time values, "greater" means "later". If the arguments are of different types, the usual conversions apply.
'ge'	Takes exactly two arguments of any type and evaluates to <code>True</code> if the first argument is greater than the second one or equal.
'lt'	Takes exactly two arguments of any type and evaluates to <code>True</code> if the first argument is lower than the second one.
'le'	Takes exactly two arguments of any type and evaluates to <code>True</code> if the first argument is lower than the second one or equal.

- 'like' Takes exactly two arguments, where the first is a string value to test, and the second is a string to match against. The second string can contain the following wildcards: ? matches any character, and % matches any number (including zero) of characters. This operation evaluates to **True** if the strings match and to **False** if they don't.
- 'notlike' Takes exactly two arguments and is the logical inversion of 'like'.
- 'between' Takes exactly three arguments and evaluates to **True** if the first argument is greater or equal than the second and lower or equal than the third one, and **False** otherwise.
- 'notbetween' Takes exactly three arguments and evaluates to **True** if the first argument is either lower than the second one or greater than the third one, and **False** otherwise.
- 'null' Takes a single argument and evaluates to **True** if it is None (NULL) and to **False** otherwise.
- 'nonnull' Takes a single argument and evaluates to **False** if it is None (NULL) and to **True** otherwise.
- 'upper' Takes a single argument of type string and evaluates to the same string with all characters converted to uppercase.
- 'lower' Takes a single argument of type string and evaluates to the same string with all characters converted to lowercase.
- 'exist' Takes at least three arguments, where the first argument is a classname, the second argument is the name of a property in the current class and the third argument is the name of a property in the class given in the first argument. The rest of the arguments can be conditions to apply to the class given in the first argument. The 'exists' operation returns **True** if and only if there exists at least one instance in the class given in argument 1 that fulfills all the conditions given in arguments 4ff and where the property given in argument 3 matches the property of the current class given in argument 2. A typical usage for this condition is to test for an instance whether it is referenced by instances of another class (in which case the third argument would usually be 'gnue\_id').