PrimeBase Systems GmbH
Max-Brauer-Allee 50
22765 Hamburg
Germany

PrimeBase Systems GmbH

PRIMEBASE

# PrimeBase™ Enterprise Objects

*An Object Oriented Framework for a new Generation of Applications*

A White Paper
by Paul McCullagh

# PrimeBase™ Enterprise Objects

*An Object Oriented Framework for a new Generation of Applications*

PrimeBase Enterprise Objects are a set of classes used to represent viewable and updateable entities for example a database table, a directory or even a text file. A PrimeBase enterprise object (PBE object) has a number of attributes used for input and output and a number of methods used for manipulating the object.

PBE objects are powerful enough to program any type of business-logic you choose. Programmers have complete control: input and output can be filtered, events and there associated actions can be defined on an object and security and access to the object can be controlled.

PrimeBase Enterprise Objects have been designed specifically with Web-based applications in mind, but may also be used in conjunction with application that supports one of the interfaces provided by the PrimeBase Virtual Machine. This includes the PrimeBase Automation Client which can use PBE Objects to route and convert data or perform periodic tasks, and the PrimeBase Open Server Remote Execution Plug-in which uses PBE Objects to perform distributed tasks.

## Table of Contents

## The Runtime Environment

PrimeBase Enterprise Objects are defined in a language called PrimeBaseTalk (PBT), and execute in the runtime environment created by the PrimeBase Virtual Machine.

PBT is a not so much a new language as a new constellation of languages. The language includes aspects of 3rd generation languages such as C and Pascal, data access components such as cursors from 4th generation languages and object oriented features from Java. The language also includes standard SQL for relational database access. All this makes PBT ideal for the expression of business logic and data manipulation tasks in general..

Once an object has been instantiated by the PrimeBase Virtual Machine (PBVM), applications and other system can communicate with these objects using any of a number of standard interfaces provided by the PBVM. These interfaces include:

- Open Database Connectivity (ODBC) the standard from Microsoft,
- Java Database Connectivity (JDBC) from Sun Microsystems and,
- Data Access Manager (DAM) and Enterprise Objects Framework Adaptor (EOF Adaptor), from Apple.

In addition to these standards, the PBVM supports 2 other open interfaces of its own.

Firstly, the PrimeBase Native Interface (PBNI) which allows PBT applications to call routines written an other languages and, the secondly PrimeBase Open Server Interface (PBOS Interface) which allows PBT applications to access any 3rd Party database server or data source using an appropriate PBOS Plug-in.

By supporting the industry standard interfaces and other open application programming interfaces (APIs) access to the PBE Objects running in this environment is pervasive and universal.

## 3-Tiered Applications

PBE Objects assumes an application that is divided into 3 layers. The upper layer is the user interface and the lower layer where the data storage takes place. The middle layer, containing the program logic, is formed by the PBE Objects.

### The User Interface

The PBE Objects do not provide any user interface functionality. In other words: the objects do not represent or draw any parts of a user interface, such as a window, or a control. However, PBE Objects can respond to input and other events generated by a user interface.

The user interface is provided by an application that then sends the various events to the PBE Objects using any of the open interfaces provided by the PrimeBase Virtual Machine mentioned above. A good example of this is the PrimeBase Application Server, which establishes the communication between the user interface provided by a standard Web browser and the PBE Objects.

Since PBE Objects program logic is independent of the user interface the re-usability of an implementation is greatly increased. The same objects can be used on multiple computer platforms and by diverse application.

### The Data Access & Storage Layer
PBE Objects are not able to store data persistently, or to control multi-user access to a single data source. This is the domain of the Data Storage Layer, which forms the foundation of a 3-tier application.

This function is usually performed by a relation database server that supports the Structured Query Language (SQL) database access language. The access to such database is native the PrimeBase Virtual Machine which provides the runtime environment for the PBE Objects.

Because of the native support, PBE Objects can access 3rd party database servers, such as Oracle, as easily as our own PrimeBase Database Server. The PBT language automatically removes any differences between SQL dialects spoken by the various database servers, truly realizing the "write once - deploy anywhere" goal needed for maximum productivity.

Today, however, data sources are much more diverse and relational databases are just one possibility. There are a vast number of resources available on the internet and a PBE object can exchange data with these libraries and archives using the PrimeBase Open Server interface provided by the PrimeBase Virtual Machine.

### The PBE Object Layer
Between the User Interface Layer the Data Storage & Access Layer is the PBE Object layer. The PBE Objects serve a number of purposes:

- They retrieve data in a form ready for display by the user interface.
- They receive data and events from the user interface and translate them into database update statements.
- The objects serve as a data cache in order to reduce the number of database operations.
- They restrict access to data depending on user privileges, and ensure that the stored information remains consistent.
- They convert and manipulate data according to the requirements of the application.

The 3 layered division of applications possible with PBE Objects is a significant advantage during development. The 3 aspects: user interface, program logic and database structure require the attention of people with quite different skills.

All these skills are often not found in one and the same person. The 3-tiered application furnishes a clear separation between these diverse tasks and provides a good basis for teamwork.

## PBE Object Types

The PBE Objects are distinct entities, each representing an updateable view of one aspect of the data. Various types, or classes, of objects are provided for use with various types of data.

### Objects, Attributes & Methods
The PBE object have a set of PBE attributes and a set of PBE methods. Each attribute represents a single value. The methods perform the operations on the objects.

The basic PBE object serves as a the basis for all other types of PBE objects. This includes PBE containers which extend PBE objects directly, and PBE tables which extend PBE containers.

PBE Objects and derived types all have a certain number of "built-in" methods and attributes. For example, the basic PBE object has 5 built-in PBE methods: `init`, `display`, `clear`, `keep` and `revert`. The basic PBE object also has a built-in attribute called `pbe_error`.

The build-in methods perform various standard functions. For example, the `revert` PBE method discards the last input value of all attributes of the object. The attributes contain data pertaining to this standard functionality. For example the built PBE attribute, `pbe_error`, contains the result code of the last method executed.

Built-in PBE attributes and methods are inherited by PBE object types that extend these objects. This means, for example, that a PBE container (and a PBE table), also have the attributes and methods of the basic PBE object. The extended types can override and replace existing built-in methods.

For example, the `clear` PBE method is used by a PBE container has the same effect, but works quite differently to the `clear` PBE method built into the basic PBE object. The `clear` PBE method of a basic object sets all PBE attributes to `$null`, on the other hand the `clear` PBE method belonging a container sets the current row pointer to zero, which has the same effect.

## Containers, Fields & Commands

PBE containers are a direct extension of the base class of PBE objects. Containers represents a so-called "rowset". A rowset is a table of information referenced by a cursor. A rowset has one or more columns. Each column having a name and a fixed length data type. A rowset can contain zero or more rows.

The columns of a container are represented by PBE field attributes. A container must have one PBE field attribute for each column of the rowset that is to be accessed by users of the container.

An important concept for containers is the "current" row. Each rowset has a "current" row. A PBE field attribute always accesses the data in the current row when setting or returning values in a container.

In addition to this, a container also has a "selected" row. The selected row is also known as the current selection. This value may vary from the current row.

To represent these values a container has the following built-in PBE attributes:

- `pbe_current_row`: This is an integer value representing the current row. The value can range from 1 to `pbe_row_count`. 0 means there is no current row.

- `pbe_selected_row`: Holds the row number of the selected row. Zero means there is no selected row.

- `pbe_row_count`: This is the total number of rows in the rowset. If it is zero, then the rowset is empty.

In addition to the built-in methods provided by the basic PBE object PBE containers have a number of methods which perform operations on the container. PBE methods that perform operations on containers are known as PBE commands. By default, PBE commands set the current row to the selected row before activation.

The built-in PBE commands are summarized in the following table:

| Command | Description |
|---|---|
| `first` | Set the selected row to the first row (row 1). |
| `last` | Set the selected row to the last row (`pbe_row_count`). |
| `next` | Set the selected row to the previous row (+1). |
| `previous` | Set the selected row to the next row (-1). |
| `new` | Inserts a new row into the rowset after the current row. |
| `save` | Copies input data into the current row |
| `delete` | Removes the current row from the rowset |
| `enter` | If there is no current row this method performs `new`, else `save`. |

*Table. Built-in PBE methods supported by containers.*

## Tables, Columns & Queries

PBE tables are derived from PBE containers. PBE tables access data stored in one or more databases tables. Because tables are derived from containers, they have a rowset. The data in the rowset is selected from the database tables.

The columns of table are represented by PBE column attributes. PBE columns are derived from PBE field attributes. The name of the PBE column must match the name of the database table column. Database table columns that do not have a corresponding PBE column cannot be accessed by the PBE table.

One table must be specified when the PBE table is declared. This is the table that will be updated by the PBE table object. Additional tables may be included in the object by adding PBE join columns.

PBE methods that operate on tables are called PBE queries. PBE queries are derived from PBE commands. A number of built-in PBE queries perform the standard database operation, including select, insert update and delete. PBE queries keep the table rowset synchronized with the data in the database table. This means, for example, if a column is updated, then the PBE query also modifies the corresponding value in the rowset. The table below is a list of built-in PBE queries.

| Query | Description |
|---|---|
| `find` | Selects data from the database based on input column data. |
| `refresh` | Performs the previous select operation. |
| `new` | Inserts a row into the database table. |
| `save` | Updates one or more rows in the table. |
| `new` | Inserts a new row into the rowset after the current row. |
| `save` | Copies input data into the current row. |

| | |
|---|---|
| `delete` | Deletes a row from the table. |

*Table. Built-in PBE queries supported by tables.*

Unlike a PBE container, which can only accept  one row of data at a time, a PBE table object can accept input data for all rows in the rowset simultaneously. This allows a table object to update multiple rows at once.

PBE tables also have one built-in attribute called `pbe_sort_list`. This is a list of columns which should be sorted when data is selected from the table.

PBE queries that modify the database table support so-called "trigger" actions. A trigger action is a special PBE action (see PBE Actions). A trigger has access to the modification image. The modification image may have a before image and an after image. The before image the state of the table row before the update takes place, and the after image is the state after the update. A trigger can perform further database operations based on the before and after image data, or modify the update by changing the after image.

## Using PBE Objects

The standard PBE Objects provide a large source of predefined functionality that is sufficient to implement the data storage and retrieval aspects of most applications. This means your application will consist mostly of object declarations. Coding of custom classes is only required for functionality particular to your application.

For this purpose, however, all classes provided by PBE Objects used for the definition of your application are extensible. This includes the objects themselves, the attributes, methods (already mentioned above) actions and events. All this ensures that an efficient solution can be found for any requirement.

To create an application using PBE Objects you will do the following:

1. *Determined the structure and nature of your data source:*
   The structure of a database, for example, which tables it has and what columns each table has must be known before you can define PBE Objects that access the data.

2. *Declare objects that represent various aspects of the data source:*
   In the case of a relational database you will declare one or more objects per table. For this purpose you will begin by using a `PBETable` object which models the relational database table.

3. *Declare PBE attributes for data components according to the requirements of the application:*
   The object that you create in step 2 to access a database table will not necessarily access all the data in the table. In this step you specify which columns of the table you require. This step may also include the declaration of columns not necessarily in the table specified in step 2, but in a related table. These are known as `PBEJoinedColumns`.

4. *Declare any additional PBE methods you have defined for use with the object:*
   PBE Objects have a complete set of standard methods that you will need updating and retrieving data from the data source. This includes searching, inserting updating and deleting data amongst other things. If you require

special functionality for your application you can define this functionality by extending an existing `PBEMethod` class, and attaching the method to your object in the method declaration.

5. *Declare events and actions for the object:*
   In this step you use standard or user defined actions to determine the behavior of your object as a result of certain events. Standard events include input or output (performed on PBE attributes), init (initialization), display (your object is displayed by the user interface) and the activation of a PBE method in general. Standard actions are provided to: filter data, check conditions, set PBE attributes, check range and format of data, call a PBE method and much more.

6. *Determine which PBE objects and methods to enable according to the user of the application on login:*
   For this purpose the standard `PBELoginObject` provides the required infrastructure.

## Phases of Operation

A PBE application cycles through various phases of operation. The first operational phase of operation is initialization. After this the application cycles through phases of input, activation and output.

The following is a description of each phase.

### The Initialization Phase

This phase only occurs once, just after all PBE objects for the application have been created. During this phase the `init` PBE method belonging to each object is activated. This is also referred to as the `init` Event. Objects may use the init event to set there initial state.

### The Input Phase

A PBE application is event driven. This means it waits for input from the user before doing anything.

When input does come, it arrives in the form of a batch of values for PBE attributes and methods belonging to various objects. Before the input batch is processed a method called `clearInput()` belonging to each of the PBE object sets the status of all PBE attributes to `$null`. This allows the application to determine later which attributes received input.

PBE methods that receive input are made "active". This means they are placed on a global activation queue held by the application. Active methods are activated in the next phase (see below).

### The Activation Phase

This phase immediately follows the input phase. The method function `activateAll()` belonging to the PBE application is called. This method calls the PBE object method function `performActivation()` for each PBE method on the active queue.

*Please note the distinction between a "PBE method" which is an object and a "method function" which is a piece of code associated with a class written in the PBT language . In this document a PBE method or a method function is sometimes simply referred to as a "method". In this case the context makes it clear which type of method is intended.*

The activation of PBE methods may trigger the activation of other methods and so on. Special "link" objects, for example, are automatically activated when the linked object is activated. This has a cascading effect.

The activation phase ends when there are no more PBE methods on the application activation queue.

### The Output Phase

In the last phase of operation, the user interface "redraws", calling the displayed PBE objects to perform output. Each object that is called to perform output may also receive a `display` event.

The user interface, however, may choose not to send the display event although attributes of a particular object are being output. In this way, the user interface can override whatever action the object normally takes on display.

Similarly to the `init` event, the display event is the activation of the `display` method. If the display method is not enable then, by default, the object will redirect the application to a different page. In this way, the display method is used to ensure against unauthorized viewing of an object.

## Application Structure

A PBE application consists of a set of PBE objects. The global value `PBE` references the global PBE application object. `PBE` is a value of type `PBEApplication` (or descendant class of `PBEApplication`). The class definition of `PBEApplication` begins as follows:

```
class PBEApplication extends WebApplication {
   public static PBETag[]        tags := new PBETag[0];
   public static PBEAction[]     actions := new PBEAction[0];
   public        PBEObject[]     objects;
```

The PBE objects are stored in array of type `PBEObjects` called `'objects'`. It also contains a list of tags and a list of actions (global to all applications) which will be discussed later.

As mentioned before, all PBE objects have a set of attributes which represent the data held by the object, and a set of methods which perform operations on the data. Each has a globally unique name.

The basic object class is the `PBEObject`. The class definition of `PBEObject` begins as follows:

```
class PBEObject extends PBEUnit {
   public      varchar           name;
   public      PBEAttribute[]    attributes;
   public      PBEMethod[]       methods;
   …
```

In the definition we define the class `PBEObject` as having a name and two arrays: '`attributes`' and '`methods`'. The first array is an array of `PBEAttribute` type objects and the second array is an array of `PBEMethod` type objects. PBE attributes and methods also have names.

As a result of this structure, each element of the application can be uniquely identified and referenced. For example, the value of the attribute '`discount`' belonging to the object '`customer`' can be retrieved as follows:

```
PBE:objects['customer']:attributes['discount']:getValue()
```

We begin with the global reference to the application , '`PBE`', and then reference the PBE object using the '`objects`' array. From the object we can select the PBE attribute using the '`attributes`' array. Note that these arrays are, so-called, associative arrays because you can address the array elements using a string value as the index. Associative values are case sensitive therefore '`customer`' is not the same as '`Customer `' in the example above.

Each object also has an array of methods, so activating the '`save`' method of the '`customer`' object can be done as follows:

```
PBE:objects['customer']:methods['save']:performActivation()
```

To find out how many attributes a PBE object has, you can use the '`length`' attribute as follows:

```
PBE:objects['customer']:attributes:length
```

You can enumerate all PBE attributes, methods or objects using an integer value has an index. Index values range from 0 to length-1. For example:

```
for (i:=0; i< PBE:objects['customer']:attributes:length; i++)
   print PBE:objects['customer']:methods[i]:name;
```

This code fragment prints out the names of the PBE attributes belonging to the PBE '`customer`' object.

## PBE Attributes

The basic PBE attribute class is the class PBEAttribute. PBEAttributes have a name and a reference to the PBE object that the attribute belongs to:

```
class PBEAttribute extends PBEUnit {
   public      objname          name;
   public      PBEObject        object;
   …
```

Each attribute holds a value. The method `getValue()` returns the current value of an attribute. The value can be assigned using the `setValue()` method.

The functionality of an attribute mainly has to do with the manipulation of the attribute value. In this section we discuss the features of attributes in this regard.

## The Edit Buffer

There is a distinction between the "edit" value and the "actual" (or current) value of an attribute. The edit value is a value held in a temporary buffer until the actual value is set. The `setValue()` method always sets the edit value, and the `getValue()` methods returns the edit value by default, if it is different to the actual value.

How this "buffering" mechanism is implemented depends on the class of the type of attribute. In basic PBEAttribute class the edit value and the actual value are one and the same value. PBEField attributes derived directly from PBEAttribute and used in conjunction with PBE container objects (see below) supports edit value buffering of one row of a container. The PBEColumn attribute used in conjunction with a PBETable object supports buffering of the entire rowset.

The purpose of the edit value is to hold a value that is being edited. It is also used to hold a value until it is suitable for replacing the current value. For example, if the user wishes to set the value of an attribute that only accepts numeric values, but inputs the value "123a" by mistake, then the value "123a" cannot replace the current attribute value immediately . Instead the new value is held in the edit buffer until the user has corrected the value.

When an edit value is valid, it is a PBE method belonging to the attribute's object that replaces the current value with the edit value for the attribute. How this is done depends on the buffering mechanism, and is PBE object type specific.

## Input and Output

Input and output refers to the exchange of data to and from the user interface. The view that a user has of an attribute value may be different to the actual value stored in the attribute. This difference is achieved by input and output "filtering".

The method `outputValue()` is used when outputting a value to the user and the `inputValue()` method is called when data is submitted by the user. These methods should never be called directly from within the application. Use `getValue()` and `setValue()` to access the value of the attributes directly in your program.

Input and output data is dispatched over the PBE object to which the attribute belongs. The input and output dispatch methods of the PBE object, therefore are the only methods that call an attributes `inputValue()` and `outputValue()` methods.

We have already mentioned data filtering. In fact, filtering is only a one possibility in any number of potential input or output event actions. Three events are defined for input and output:

- `Before (or On) Input`: This event occurs before a value is input, and can be used (amongst other things) to filter the input data.
- `After Input`: This event occurs after input.
- `Before (or On) Output`: This event occurs before a value is output and can be used to filter the output value.

For each event the attribute stores an executable string. These strings are "compiled" out of events and actions specified for the PBE attribute when the object is defined.

### The Attribute Status

The attribute status indicates whether an attribute received input during the last input phase, and whether an error occurred during input.

The status is a string (`varchar`) value which has the following meaning:

> *An empty string ("")*: This means a value has been input and no error occurred.
>
> *A NULL string* *(`$null`)*: This means no value was input.
>
> *A non-empty string*: This means a value has been input, but the value is not valid. The string value specified the error.

The `getStatus()` and `setStatus()` methods are used to get and set the status value of the attribute.

## PBE Methods

The PBEMethod class is derived from PBEAttribute. Methods perform the actual operations in a PBE application. All PBE methods, include PBE commands  and queries are derived from the PBEMethod class.

### Activation

Since `PBEMethod` is derived from `PBEAttribute` methods have a value, and can receive input just like an attribute. When a method receives input it becomes active, and places itself on the application active queue. During the activation phase, the method will be activated by the application.

To activate a PBE method, the application calls the `performActivation()` method function belonging to the PBE method. The `performActivation()` method calls the `activate()` method to do the actual work of the activation.

`performActivation()` initially sets the method status to `""`. The PBE method status is derived from the attribute status and is therefore string (`varchar`) value. The meaning of the status value after execution is as follows:

> *An empty string ("")*: This means the methods was activated successfully.
>
> *A NULL string* *(`$null`)*: This means the method was not activated, however no error occurred. For example, this happens when the methods is disabled.
>
> *A non-empty string*: This means an error occurred activation was not completed. In this case the value contains a description of the error.

If an error  occurs during activation, then the error is passed on to the activating PBE object (see below). If no error occurs, then the PBE object will clear the input data.

### Events

There are two type of events supported by all PBE methods:

- `Before (or On) Activation`: This event occurs the method is activated. It can be used, for example, to setup or change data for the actual activation.

- `After Activation`: This event occurs after activation. This event only occurs if the activation completed successfully. It is useful for triggering actions that need to always occur after a particular method has executed. For example, redirecting to a page containing a list after a `find` PBE method was activated.

Just like input and output events, before and after events are stored as executable strings by a PBE method. The strings are compiled from specifications made when the objects are defined.

### Enable & Disable

A PBE method can be either enabled or disable. If it is disable, the method will not activate. The of the attribute will always be set to `""` and `performActivation()` returns `$false`. Before or after events are also not executed if a method is disabled.

On startup, all methods are disabled by the application. When a user logs in, depending on whether the user is a guest, super-user or normal user, methods are enabled accordingly.

## PBE Objects

The most basic PBE class is the `PBEUnit`. This class provides methods for compiling event descriptions. The actual events are handled by the `PBEObject` and `PBEAttributes` type objects which are directly descended from `PBEUnit`.

The basic object class is the `PBEObject`. All PBE objects are descended from this class. Besides an array of attributes and methods, the PBEObject also includes method functions which do the following:

- control access to the objects (`redirect()`, `enable()` and `disable()`),
- handle activation (`setActive()` and `performActivation()`),
- set and get the object error status (`setError()` and `getError()`)
- dispatch input and output to and from the attributes (`inputValue()` and `outputValue()`),
- dispatch input to the methods (`inputMethod()`),
- dispatch status information to and from the attributes (`getAttributeStatus()` and `setAttributeStatus()`).

Note that all exchanges between the user-interface and the object attributes are dispatched by methods belonging to the object. This dispatch methods are given the name of the attribute and other parameters as required.

The dispatch mechanism allows the object to examine the name given and transform or interpret the name depending on the object type. This application of the dispatch mechanism becomes more obvious as examine the more sophisticated objects below.

All objects have one built-in attribute called '`pbe_error`'. This attribute contains the result of the last method activated (executed). The result is string (`varchar`) and is derived from the status of the activated method.

In addition to this, all objects have 5 built-in methods:

> `init`: This method is activated when the application initializes. The method is always enabled (cannot be disabled). The default `init` method does nothing. In general, the `init` method is intended as an event hook for actions your application should perform during initialization.
>
> `display`: This method is activated when the object is displayed by the user-interface. If the `display` method is not activated, then it will re-direct to a page specified by the value stored by the method.
>
> `clear`: This method sets all attributes of the object to `$null`. `$null` represents a missing value.
>
> `keep`: Normally the input data is cleared after successful execution. The `keep` method preserves and holds the input data for the future activation of some other method. In this sense, the keep method serves a place holder that is equivalent to performing no operation.
>
> `revert`: This method discard the input data. If input data is buffered, as it is in PBE containers and tables, then the fields and columns values will be returned to there pre-input state.

## PBE Actions

PBE actions perform the work during an event. Actions can do such things as set a PBE attribute value or status, enable, disable or activate a PBE method, cause redirection to a different page in the application, and much more. It is therefore possible to program a large part of your application logic using actions which caused by events.

All PBE actions are derived from the class `PBEAction`. All actions have a method function called `perform()` which does the work of the action.

`perform()` accepts a number of parameters. The first two are standard, and include:

- A reference to the PBE method that triggered the action. In the case of input and output events no PBE method is involved, and therefore this parameter is set to `$null`.
- A reference to the PBE object or attribute to which the event was originally attached.

Subsequent input parameters are specific to the action type.

### Action Types

The type of work an action does depends in the class of the action. The following types of actions are available:

- `PBEAction`: These are generic actions that may be applied to any type of object or attribute.

- `PBEAttributeAction`: Attribute actions may only be used in events placed on PBE attributes.

- `PBECondition`: A condition is an action that returns either TRUE or FALSE. It is used to limit that application of ather actions in the same event.

- `PBEAttributeCondition`: This is a condition that can only be applied to attribute events.

- PBEFilter: A filter is an attribute action. It can only be applied before input and output. A filter is used to transform, convert or map attribute values during input and output. One of the input parameters of the `perform()` method used by a filter is always the value of the attribute. The modified value is returned by the filter.

- PBETrigger: This is a special type of action that can only be applied to PBE table objects. A trigger may be used to modify the data a PBE query. It can also be used to make further updates based on how the PBE query modifies the PBE table rowset.

## Standard Actions

PBE Objects offers a large number of standard actions for performing common tasks. In addition to actions that may be applied to any PBE object or attribute there are specialized actions which only work on certain advanced classes of PBE objects or attributes. For example, the 'setmax' action requires a PBE container.

The following table summarizes standard actions provided by PBE Objects:

| Action | Description |
|---|---|
| `do` | Activate a given PBE method. |
| `enable` | Enable a PBE method. |
| `goto` | Transfer control to a different page. |
| `exec` | Compile a run a code string. |
| `set` | Set a PBE attribute to a given value. |
| `seterror` | Set the error status of a PBE object or status of a PBE attribute. |
| `use` | Indicate that an attribute is to be used during activation of a method. |
| `required` | Indicate that an attribute value is required for activation of a method . |
| `ignore` | Indicate that the value of an attribute should be ignored for activation. |
| `range` | Specify a number range for an attribute. |
| `test` | Test any given condition. |
| `error` | Test if an error has occurred on a PBE object or attribute. |

| | |
|---|---|
| `page` | Test if the current page is a particular page. |
| `empty` | Test of an attribute is empty ($null or empty string). |
| `format` | Specify the form of a PBE attribute value. |
| `map` | Specify a mapping for a PBE attribute value . |
| `embed` | Embed an attribute value is a given string on output . |
| `fetch` | Set the selected row of a PBE container . |
| `selected` | Test the selected row of a container. |
| `setmax` | Set a PBE attribute to the maximum value of a container field. |
| `unique` | Indicate that a table column must be unique. |

*Table. Standard actions provided by PBE Objects.*

## Declaring Objects & Events

A PBE application is created by declaring PBE objects and their associated events. For this purpose, the PBE application provides "factory" method functions that create object instances as they are declared. The method `declareObject()` is used to declare a PBE object, and the method `declareAttribute()` is used to declare an attribute. For example:

```
PBE:declareObject("PBETable", "customer", "Customers", "num");
PBE:declareAttribute("PBEColumn", "num");
PBE:declareAttribute("PBEColumn", "name");
PBE:declareAttribute("PBEColumn", "address");
PBE:declareAttribute("PBEColumn", "discount");
PBE:declareAttribute("PBEColumn", "country);
```

In the above example an object called customer with 5 attributes is declared. PBE attributes must always be declared immediately following the object declaration..

The first parameter of the factory methods is the type (class) of the object being declared. For example, for `declareObject()`: `PBEObject`, `PBEContainer`, `PBETable`, `PBELinkTable`, etc. or for `declareAttribute()`: `PBEAttribute`. `PBEField`, `PBEColumn`, etc.

The second parameter is the name of the PBE object or attribute. Parameters that follow depend on the class of the object or attribute. PBETable, for example, requires the name of the database table as third parameter and then name of the table key column(s) as the 4th parameter.

Just like attributes, PBE methods can also be added to an object declaration using the `declareMethod()` factory method, for example:

```
PBE:declareMethod("ReportSalesMethod", "report");
```

After declaration of an object, the events for the object can be declared. Events can be declared for PBE objects or attributes using either the `objectEvent()` or the `attributeEvent()` method functions. For example:

```
PBE:objectEvent("customer", "after:find:goto(custlist.lml)");
PBE:attributeEvent("customer", "name", "on:new,save:required");
```

This first parameter is the name of the PBE object. In the case of `attributeEvent()`, the second parameter is the name of the attribute on which the event is placed. Following this is the specification of the event. The first event specifies that after the `find` PBE method is activated, the user must be directed to the '`custlist.lml`' page. The second event specifies that when the `new` or `save` PBE methods are activated, then a value for the `name` attribute is required.

In general, event specifications have the following format:

```
   [when]: [what]:[action sequence]
```

`[when]` is either '`before`', '`on`' or '`after`', whereby '`before`' and '`on`' have basically the same meaning. `[what]` can be either '`input`' or '`output`', or the name of a PBE method. This includes the methods `init` and `display`.

The `[action sequence]` is a series of actions. Each action is specified by the action name optionally followed by a 1 or more parameters. Actions are executed in the order in which they appear in the sequence. Several actions may be separated by a space.

Condition actions must be prefixed by '`if`' or '`if not`'. Depending on the result of the condition action, the actions following the condition will be either executed or not. For example:

```
PBE:attributeEvent("customer", "discount", "on:new:if empty
set(10)");
```

This event checks value of the discount PBE column, and sets the value to 10 if no value has been given. The event is executed just be the `new` PBE method belonging to the '`customer`' object is activated.

## PBE Tags

PBE tags are an implementation of LiveTags™ for PrimeBase Enterprise Objects. PBE tags are therefore the link between the Web-based user interface and the PBE Objects environment. In this implementation of LiveTags, each tag is represented by an object.

The basic PBE tag has a `generate()` method and a `generateEnd()` method function. `generate()` creates the start HTML tag, and `generateEnd()` creates the end tag if necessary. Calls to `generate()` and `generateEnd()` methods are dispatched from the PBE application methods `liveTag()` and `liveTagEnd()` and the more specialized `selectTag()` and `selectTagEnd()` methods. The mapping from HTML tags to call to these methods are specified in the PrimeBase Application Server tagmap file 'pbetags.map'.

Input and output to and from the HTML based user interface occurs via PBE tags. A specialization called a `PBEOutputTag` performs output, while a `PBEInputTag` can perform both input and output.

Let us consider the following HTML code:

```
<form name="customer">
   <input type="text" name="address" size=40 value="">
</form>
```

The name of the `<form>` tag ("customer ") refers to a PBE object, and the name of the `<input>` tag ("address ") refers to a PBE attribute belonging to that object. In the generated tag, the value attribute belonging to the `<input>` tag (value="" above) must be set to the current value of the referenced PBE attribute.

The following is a description of what happens when displaying the `<input>` tag in following HTML:

1. According to the LiveTag map, '`pbetag.map`', The application server makes a call to the `liveTag()` method belonging to the PBE application, which includes the name of the tag, the name of the enclosing `<form>` tag and the name of the `<input>` tag:

   ```
   PBE:liveTag("text", "customer", "address", "40",
   …)
   ```

2. The application uses the HTML form name (in this case "customer") and searches for a PBE object with same name. It does this by using the form name as an index into the array of PBE objects held by the application:

   ```
   PBEObject object := this:objects["customer"];
   ```

   The PBE application then uses the name of the tag to find the associated tag object. This is done by using the tag name as an index into the global tag array held by the application.

   ```
   PBETag tag := this:tags["text"];
   ```

   This information is then passed on in a call the tag in a call to the `generate()` method of the tag:

   ```
   tag:generate(object, "customer", 40, …);
   ```

3. The `input` PBE tag object will generate an input tag, but first it needs to know the current value of the referenced PBE attribute. For this purpose it requests output from the PBE object, passing the name of the PBE:

   value := object:outputValue("customer");

4. The object now takes control. It uses the name given as an index into its array of PBE attributes, and then dispatches the output call to the attribute. It collects and returns the resulting value to the tag:

   ```
   PBEAttribute attr := this:attributes["customer"];
   value := attr:outputValue();
   ```

5. The `input` PBE tag then generates the new HTML `<input>` tag with the HTML tag attribute `value` set to the correct value:

   ```
   <input type="text" name="address" size=40
   value="1234 52nd Street">
   ```

## Conclusion

PrimeBase Enterprise Objects represent a powerful new way of creating applications. Existing components can be combined and assembled to form new components that are themselves reusable. The standard classes provide a great deal of ready-to-use functionality which can also be easily extended.

Seamless integration with Web based user interfaces and database back-ends is a major feature of PBE Objects, and it is this fact that gives the system an even greater potential. PBE Objects are built to serve program logic to the internet or intranet and this, we believe, is the future of application programming.