Interchange Ecommerce Functions

Table of Contents

1. THE ORDER PROCESS

Interchange has a completely flexible order basket and checkout scheme. The `foundation` demo presents a common use of this process, in the directory pages/ord −− the files are:

```
basket.html      The order basket displayed by default
checkout.html    The form where the customer enters their billing
                 and shipping info
```

and in the directory etc:

```
receipt.html     The receipt displayed to the customer
report           The order report mailed to you
mail_receipt     The customer's email copy (if requested)
```

It is not strictly necessary to display an order basket when an item is ordered. If you specify a different page to be displayed that is fine, but most customers will be confused if you don't give them an indication that the order operation has succeeded.

Any order basket is an HTML `FORM`. It will have a number of variables on it. At the minimum it must have an `[item−list]` to loop through the items, and the `quantity` of each item must be set in some place on that form. Any valid Interchange tags may be used on the page, and you may use multiple item lists if necessary.

1.1. How to order an item

Interchange can either use a form−based order or a link−based order to place an item in the shopping cart. The link−based order uses the special `[order item−code]` tag:

[order code]

named attributes:

```
[order code="sku" quantity="n"* href="page"* cart="cartname"* base="table"*]
* = optional paramenters
```

Expands into a hypertext link which will include the specified code in the list of products to order and display the order page. **code** should be a product SKU listed in one of the "products" tables, and is the only required parameter. **quantity** may be specified if more than one (the default) of the item should be placed in the cart. **href** allows some page other than the default order page to be displayed once the item has been added to the cart. **cart** selects the shopping cart the item will be placed in. The optional argument **base** constrains the order to a particular products file −− if not specified, all tables defined as products files will be searched in sequence for the item.

Example:

```
Order a [order TK112]Toaster[/order] today.
```

Note that this is the same as:

```
Order a [page order TK112]Toaster</A> today.
```

You can change frames for the order with:

```
Order a <A HREF="[area order TK112]" TARGET=newframe>Toaster</A> today.
```

[/order]

Expands into . Used with the order element, such as: Buy a `[order TK112]`Toaster`[/order]` today.

To order with a form, you set the form variable `mv_order_item` to the item−code/SKU and use the `refresh` action:

```
<FORM ACTION="[process−target]" METHOD=POST>
<INPUT TYPE=hidden  NAME="mv_todo"        VALUE="refresh">
<INPUT TYPE=hidden  NAME="mv_order_item"  VALUE="TK112">

Order <INPUT NAME="mv_order_quantity" SIZE=3 VALUE=1> toaster

<INPUT TYPE=submit VALUE="Order!">
</FORM>
```

You may batch select whole groups of items:

```
<FORM ACTION="[process−target]" METHOD=POST>
<INPUT TYPE=hidden  NAME="mv_todo"        VALUE="refresh">

<INPUT TYPE=hidden  NAME="mv_order_item"  VALUE="TK112">
<INPUT NAME="mv_order_quantity" SIZE=3> Standard Toaster

<INPUT TYPE=hidden  NAME="mv_order_item"  VALUE="TK200">
<INPUT NAME="mv_order_quantity" SIZE=3> Super Toaster

<INPUT TYPE=submit VALUE="Order!">
</FORM>
```

Items that have a quantity of zero (or blank) will be skipped, and only items with a positive quantity will be placed in the basket.

You may also specify attributes like size or color at time of order (see *How to set up an order button*).

1.2. How to set up an order link

On a product display page, use:

```
[order 00−0011]Order the Mona Lisa[/order]
```

If coming from a search results or on−the−fly page, you may use the generated `[item−code]` thusly:

```
[order [item−code]]Order [item−field name][/order]
```

Bear in mind that if you have not reached the page via a search or on−the−fly operation, `[item−code]` means nothing and will cause an error.

1.3. How to set up an order button

Interchange can order via form submission as well. This allows you to order a product (or group of products) via a form button. In its simplest form, it is:

```
<FORM ACTION="[process-target]" METHOD=POST>
<INPUT TYPE=hidden NAME=mv_todo VALUE=refresh>
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011">
<INPUT TYPE=submit VALUE="Order the Mona Lisa">
</FORM>
```

The default quantity is one. An initial quantity may be set by the user by adding an mv_order_quantity variable:

```
 Number to order:<INPUT TYPE=text NAME=mv_order_quantity VALUE="1">
```

You can order multiple items by stacking the variables:

```
<FORM ACTION="[process-target]" METHOD=POST>
<INPUT TYPE=hidden NAME=mv_todo VALUE=refresh>
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011a">
<INPUT TYPE=submit VALUE="Order the Mona Lisa with frame">
</FORM>
```

Initial size or color may be set as well, provided *UseModifier* is set up properly:

```
<INPUT TYPE=hidden NAME=mv_order_size VALUE="L">
```

If the order is coming from a generated flypage, loop list, or search results page, you can get a canned select box from the `[item-accessories size]` or `[item-accessories size]` tag. See *Item Attributes*.

1.4. How to set up an on-the-fly item

If you enable the catalog directive *OnFly*, setting it to the name of a subroutine (or possibly a UserTag) that can handle its calls, then Interchange will add items to the basket that are not in the product database. Interchange supplies an internal `onfly` subroutine, which will work according to the examples given below.

In `catalog.cfg`:

```
OnFly  onfly
```

If your item code is not to be named `mv_order_item` then you must perform a rename in the `Autoload` routine.

A basic link can be generated like:

```
<a href="[area form="
        mv_todo=refresh
        mv_order_item=000101
        mv_order_fly=description=An on-the-fly item|price=100.01
"]">Order item 000101</a>
```

The form parameter value `mv_order_fly` can contain any number of fields which will set corresponding parameters in the item attributes. The fields are separated by the pipe (`|`) character and contain value−parmeter pairs separated by an = sign. (These are URL−encoded by the `[area ...]` or `[page ...]` tag, of course.) You can set a size, color, or any other parameter.

The special attribute `mv_price` can be used in conjunction with the `CommonAdjust` atom `$` to set the price for checkout and display.

The `[item−list]` sub−tag `[item−description]`, when used with an item−list, will use the item attribute `description` to display in the basket. Note that `[item−field description]` or `[item−data products description]` will NOT work, as both of these tags reference an actual field value for a record in the products table − not applicable for on−the−fly items. Similarly, an attempt to generate a flypage for an on−the−fly item (`[page 000101]`, for example) will fail, resulting in the display of the SpecialPage **missing**.

If you wish to set up a UserTag to process on−the−fly items, it should accept a call of

```
usertag(mv_item_code, mv_item_quantity, mv_order_fly)
```

The `mv_item_code` and `mv_order_fly` parameters are required to trigger Interchange's `add_item` routine (along with mv_todo=refresh to set the action).

The item will always act as if `SeparateItems` or `mv_separate_items` is set.

Multiple items can be ordered at once by stacking the variables. If there is only one `mv_order_item` instance, however, you can stack the `mv_order_fly` variable so that all are concatenated together as with the `|` symbol. So the above example could be done as:

```
[area form="
        mv_todo=refresh
        mv_order_item=000101
        mv_order_fly=description=An on−the−fly item
        mv_order_fly=price=100.00
"]
```

Multiple items would need multiple instances of `mv_order_item` with a corresponding `mv_order_fly` for each `mv_order_item`. You can order both `000101` and `000101` as follows:

```
[area form="
    mv_todo=refresh

    mv_order_item=000101
    mv_order_fly=description=An on−the−fly item|price=100.00

    mv_order_item=000102
    mv_order_fly=description=Another on−the−fly item|price=200.00
"]
```

The following two forms correspond to the above two examples, in order, with the slight refinement of adding a quantity:

```
<FORM ACTION="[area process]" METHOD=POST>
      <INPUT TYPE=hidden NAME=mv_todo VALUE="refresh">
      <INPUT TYPE=hidden NAME=mv_order_item VALUE="000101">
```

```
    Qty: <INPUT SIZE=2 NAME=mv_order_quantity VALUE="1">
    <INPUT TYPE=hidden NAME=mv_order_fly
            VALUE="description=An on-the-fly item|price=100.00">
    <INPUT TYPE=submit VALUE="Order button">
</FORM>

<FORM ACTION="[area process]" METHOD=POST>
    <INPUT TYPE=hidden NAME=mv_todo VALUE="refresh">
    <INPUT TYPE=hidden NAME=mv_order_item VALUE="000101">
    Qty: <INPUT SIZE=2 NAME=mv_order_quantity VALUE="1"><BR>
    <INPUT TYPE=hidden NAME=mv_order_fly
        VALUE="description=An on-the-fly item|price=100.00">
    <INPUT TYPE=hidden NAME=mv_order_item VALUE="000102">
    Qty: <INPUT SIZE=2 NAME=mv_order_quantity VALUE="1"><BR>
    <INPUT TYPE=hidden NAME=mv_order_fly
        VALUE="description=Another on-the-fly item|price=200.00">
    <INPUT TYPE=submit VALUE="Order two different with a button">
</FORM>
```

1.5. Order Groups

Interchange allows you to group items together, making a master item and sub−items. This can be used to delete accessories or options when the master item is deleted. In its simplest form, you order just one master item and all subsequent items are sub−items.

```
<FORM ACTION="[process-target]" METHOD=POST>
<INPUT TYPE=hidden NAME=mv_todo VALUE=refresh>
<INPUT TYPE=hidden NAME=mv_order_group VALUE="1">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011a">
<INPUT TYPE=submit VALUE="Order the Mona Lisa with frame">
</FORM>
```

If you wish to stack more than one master item, then you must define mv_order_group for **all** items, with either a 1 value (master) or 0 value (sub−item). A master owns all subsequent sub−items until the next master is defined.

```
<FORM ACTION="[process-target]" METHOD=POST>
<INPUT TYPE=hidden NAME=mv_todo VALUE=refresh>
<INPUT TYPE=hidden NAME=mv_order_group VALUE="1">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011">
<INPUT TYPE=hidden NAME=mv_order_group VALUE="0">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="00-0011a">
<INPUT TYPE=hidden NAME=mv_order_group VALUE="1">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="19-202">
<INPUT TYPE=hidden NAME=mv_order_group VALUE="0">
<INPUT TYPE=hidden NAME=mv_order_item VALUE="99-102">
<INPUT TYPE=submit VALUE="Order items">
</FORM>
```

When the master item `00-0011` is deleted from the basket, `00-0011a` will be deleted as well. And when 19−202 is deleted, then 99−102 will be deleted from the basket.

NOTE: Use of checkboxes for this type of thing can be hazardous, as they do not pass a value when unchecked. It is preferable to use radio groups or select/drop−down widgets. If you must use checkboxes, be sure to explicitly clear `mv_order_group` and `mv_order_item` somewhere on the page which contains the form:

```
[value name=mv_order_group set='']
[value name=mv_order_item set='']
```

The attributes `mv_mi` and `mv_si` are set to the group and sub−item status of each item. The group, contained in the attribute `mv_mi`, is a meaningless yet unique integer. All items in a group will have the same value of `mv_mi`. The attribute `mv_si` is set to 0 if the item is a master item, and 1 if it is a sub−item.

1.6. Basket display

The basket page(s) are where the items are tracked and adjusted by the customer. It is possible to have an unlimited number of basket pages. It is also possible to have multiple shopping carts, as in buy or sell. This allows a basket/checkout type of ordering scheme, with custom order pages for items which have many accessories.

The name of the page to display can be configured in several ways:

1. Set the SpecialPage `order` to the page to display when an item is ordered.
2. Use the `[order code=item page=page_name] Order it! [/order]` form of order tag to specify an arbitrary order page for an item.
3. If already on an order page, set the mv_orderpage, mv_nextpage, mv_successpage, or mv_failpage variables.

The following variables can be used to control cart selection and page display:

mv_cartname

The shopping cart (default is main) to be used for this order operation.

mv_failpage

Page to be displayed on a failed order check (see *Advanced Multi−level Order Pages*)

mv_nextpage

Page to display on a return operation.

mv_orderpage

Page to be displayed on a refresh.

mv_successpage

Page to be displayed on a successful order check (see *Advanced Multi−level Order Pages*).

mv_order_profile

Order profile to be used if the form action is `submit` (see *Advanced Multi−level Order Pages*).

1.7. Multiple Shopping Carts

Interchange allows you to define and maintain multiple shopping carts. One shopping cart −− main, by name −− is defined when the user session starts. If the user orders item M1212 with the following tag:

```
[order code=M1212 cart=layaway] Order this item! [/order]
```

the order will be placed in the cart named *layaway*. However, by default you won't see the just−ordered item on the basket page. That is because the default shopping basket displays the contents of the 'main' cart only. So copy the default basket page (pages/ord/basket.html in the demo) to a new file, insert a `[cart layaway]` tag, and specify it as the target page in your `[order]` tag:

```
[order code=M1212 cart=layaway page=ord/lay_basket] Order this item! [/order]
```

Now the contents of the *layaway* cart will be displayed. Most of the ITL tags that are fundamental to cart display accept a 'cartname' option, allowing you to specify which cart to be used:

[cart cartname]

A 'sticky' setting of the default cart name to use for all subsequent cart−related tags. Convenient, but you must remember to use `[cart main]` to get back to the primary cart! As an alternative, you can specify the desired cart as a parameter of the other tags. These are not sticky, referencing the specified cart only for the instance in which they are called:

[item−list cartname]...[/item−list]

Iterates over the items in the specified cart − tags like `[item−quantity]` and `[item−price]` will be evaluated accordingly;

[nitems cartname]

Returns the total number of items in the specified cart;

[subtotal cartname]

Returns the monetary subtotal for the contents of specified cart;

[shipping cartname], [handling cartname], [salestax cartname], [total−cost cartname]

You get the idea. It is worth noting that tags which summarize cart contents do not need to be in used concert, or in conjunction with an `[item−list]`. For instance, you can display just the grand total for a cart on the sidebar or bottom of each page, using `[total−cost]` by itself, if you wish.

You can also order items from a form, using the `mv_order_item`, `mv_cartname`, and optional `mv_order_quantity` variables.

```
<FORM METHOD=POST ACTION="[process−order]">
<input type=checkbox name="mv_order_item" value="M3243"> Item M3243
<input name="mv_order_quantity" value="1"> Quantity
<input type=hidden name="mv_cartname" value="layaway">
<input type=hidden name="mv_doit" value="refresh">
<input type=submit name="mv_junk" value="Place on Layaway Now!">
```

```
</FORM>
```

If you need to utilize an alternative item price in conjunction with the use of a custom cart, see the section on *PRODUCT PRICING* for pricing methods and strategies.

2. PRODUCT PRICING

Interchange maintains a price in its database for every product. The price field is the one required field in the product database −− it is necessary to build the price routines.

For speed, Interchange builds the code that is used to determine a product's price at catalog configuration time. If you choose to change a directive that affects product pricing you must reconfigure the catalog.

2.1. Simple pricing

The simplest method is flat pricing based on a fixed value in the `products` database. If you put that price in a field named `price`, you don't need to do more. If you want to change pricing based on quantity, size, color or other factors read on.

2.2. Price Maintenance with CommonAdjust

As of Interchange 3.11, a flexible chained pricing scheme is available when the *CommonAdjust* directive is set.

NOTE: For compatibility with older carts, if both PriceAdjustment and CommonAdjust are set, and CommonAdjust contains a valid database identifier, the CommonAdjust value is used to set pricing adjustments based on item attributes. This is not discussed further in this section; all items below assume *PriceAdjustment* is not in use.

We talk below about a *CommonAdjust string*; it will be defined in due time.

A few rules about CommonAdjust, all assuming the *PriceField* directive is set to `price`:

1

If `CommonAdjust` is set to any value, a valid *CommonAdjust string* or not, extended price adjustments are enabled. It may also hold the default pricing scheme.

2

The `price` field may also hold a *CommonAdjust string*. It takes precedence over the default.

3

If the value of the `CommonAdjust` directive is set to a CommonAdjust string, and the `price` field is empty or specifically *0*, then it will be used to set the price of the items.

4

If PriceBreaks is in use, it's price will take precedence over the value of `CommonAdjust`, though it may also contain a CommonAdjust string.

5

If no CommonAdjust strings are found, then the price will be 0, subject to any later application of discounts.

6

If another CommonAdjust string is found as the result of an operation, it will be re−parsed and the result applied. Chaining is retained; a fallback may be passed and will take effect.

Prices may be adjusted in several ways, and the individual actions are referred to below as *atoms*. Price atoms they may be *final, chained,* or *fallback.* A final price atom is always applied if it does not evaluate to zero. A chained price atom is subject to further adjustment. A fallback price atom is skipped if a previous chained price was not zero.

Atoms are separated by whitespace, and may be quoted (although there should not normally be whitespace in a setting). A chained item ends with a comma. A fallback item has a leading semi−colon. Final atoms have no comma appended or semi−colon prepended.

A *settor* is the means by which the price is set. There are There are eight different types of price settors. All settors can then yield another CommonAdjust string.

It is quite possible to create endless loops, so the maximum number of initial CommonAdjust strings is set to 16, and there may be only 20 iterations before the price will return zero on an error.

NOTE: Common needs are easily shown but not so easily explained; skip to the examples if the reference below if your vision starts to blur when reading the next section. 8−)

USAGE: Optional items below have asterisks appended. The asterisk should not be used in the actual string. Optional **base** or **table** always defaults to the active `products` database table. The optional **key** defaults to the item code except in a special case for the attribute−based lookup. The **field** name is not optional except in the case of an attribute lookup.

N.NN or −N.NN

where N is a digit. A number which is applied directly; for instance 10 will yield a price of 10. May be a positive or negative number.

N.NN%

where N is a digit. A number which is applied as a percentage of the *current* price value. May be a positive or negative number. For example, if the price is 10 and −8% is applied, the next price value will be 9.20.

table*:column:key*

Causes a straight lookup in a database table. The optional **table** defaults to the main products database table for the item (subject of course to multiple product files). The **column** must always be present. The optional **key** defaults to the item code except in a special case for the attribute−based lookup. The return value is then re−parsed as another price settor.

table*:col1..col5,col10:key*

Causes a quantity lookup in database table **table** (which defaults to the products database), with a set of comma−separated fields, looked up by the optional **key**. (Key defaults to the item code, of course). If ranges

are specified with .., each column in the sequence will be used; Therefore

```
pricing:p1,p2,p3,p4,p5,p10:
```

is the same as

```
pricing:p1..p5,p10:
```

Leading non−digits are stripped, and the item quantity is compared with the numerical portion of the column name. The price is set to the value of the database column (numeric portion) that is at least equal to it but doesn't yet reach the next break.
WARNING: If the field at the appropriate quantity level is blank, a zero cost will be returned from the atom. It is important to have all columns populated.

==attribute:table*:column*:key*

Does an attribute−based adjustment. The attribute is looked up in the database **table**, with the optional **column** defaulting to the same name as the *value* of the **attribute**. If the column is not left blank, the *key* is set to the *value* of the **attribute** if blank.

& CODE

The leading & sign is stripped and the code is passed to the equivalent of a `[calc]` tag. No Interchange tags can be used, but the &tag_data routine is available, the current value of the price and quantity are available as `$s`, and the current item (code, quantity, price, and any attributes) are available as `$item`, all forced to the package Vend::Interpolate. That means that in a UserTag:

```
$Vend::Interpolate::item          is the current item
$Vend::Interpolate::item->{code}  gives key for current item
$Vend::Interpolate::item->{size}  gives size for current item (if there)
$Vend::Interpolate::item->{mv_ib} gives database ordered from
```

[valid minivend tags]

If the settor begins with a square bracket (`[`) or underscore, it is parsed for Interchange tags with variable substitution (but no Locale substitution). You may define a price in a *Variable* in this fashion. The string is re−submitted as an atom, so it may yield yet another settor.

$

Tells Interchange to look in the `mv_price` attribute of the shopping cart, and apply that price as the final price, if it exists. The attribute must be a numerical value.

>>word

Tells the routine to return `word` directly as the result. This is not useful in pricing, as it will evaluate to zero. But when CommonAdjust is used for shipping, it is a way of re−directing shipping modes.

word

The value of `word`, which must not match any of the other settors, is available as a key for the next lookup (only). If the next settor is a database lookup, and it contains a dollar sign (`$`) the `word` will be substituted;

i.e. `table:column:$` becomes `table:column:word`.

(settor)

The value returned by `settor` will be used as a key for the next lookup, as above.

2.3. CommonAdjust Examples

Most examples below use an outboard database table named **pricing**, but any valid table including the **products** table can be used. We will refer to this **pricing** table:

```
code    common  q1      q5      q10     XL      S       red
99-102          10      9       8       1       -0.50   0.75
00-343                                  2
red       0.75
```

The simplest case is a straight lookup on an attribute; *size* in this case.

```
10.00, ==size:pricing
```

With this value in the `price` field, a base price of 10.00 will be adjusted with the value of the *size* attribute. If size for the item 99−102 is set to `XL` then 1.00 will be added for a total price of 11.00; if it is `S` then .50 will be subtracted for a total price of 9.50; for any other value of *size* no further adjustment would be made. 00−343 would be adjusted up 2.00 only for *XL*.

```
10.00, ==size:pricing, ==color:pricing
```

This is the same as above, except both size and color are adjusted for. A color value of red for item code 99−102 would add 0.75 to the price. For 00−343 it would have no effect.

```
10.00, ==size:pricing, ==color:pricing:common
```

Here price is set based on a common column, keyed by the value of the color attribute. Any item with a color value of red would have 0.75 added to the base price.

```
pricing:q1,q5,q10:, ;10.00, ==size:pricing, ==color:pricing:common
```

Here is a quantity price lookup, with a fallback price setting. If there is a valid price found at the quantity of 1, 5, or 10, depending on item quantity, then it will be used. The fallback of 10.00 only applies if no non−zero/non−blank price was found at the quantity lookup. In either case, size/color adjustment is applied.

```
pricing:q1,q5,q10:, ;10.00 ==size:pricing, ==color:pricing:common
```

Removing the comma from the end of the fallback string stops color/size lookup if it reaches that point. If a quantity price was found, then size and color are chained.

```
pricing:q1,q5,q10:, ;products:list_price, ==size:pricing, ==color:pricing
```

The value of the database column `list_price` is used as a fallback instead of the fixed 10.00 value. The above value might be a nice one to use as the default for a typical retail catalog that has items with colors and sizes.

2.4. PriceBreaks, discounts, and PriceAdjustment

There are several ways that Interchange can modify the price of a product during normal catalog operation. Several of them require that the *pricing.asc* file be present, and that you define a pricing database. You do that by placing the following directive in *catalog.cfg*:

```
Database  pricing pricing.asc 1
```

NOTE: PriceAdjustment is slightly deprecated by CommonAdjust, but will remain in use at least through the end of Version 3 of Interchange.

Configurable directives and tags with regard to pricing:

- Quantity price breaks are configured by means of the *PriceBreaks* and *MixMatch* directives. They require a field named specifically `price` in the pricing database. The **price** field contains a space−separated list of prices that correspond to the quantity levels defined in the `PriceBreaks` directive. If quantity is to be applied to all items in the shopping cart (as opposed to quantity of just that item) then the *MixMatch* directive should be set to **Yes**.
- Individual line−item prices can be adjusted according to the value of their attributes. See *PriceAdjustment* and *CommonAdjust*. The pricing database **must** be defined unless you define the `CommonAdjust` behavior.
- Product discounts for individual products, specific product codes, all products, or the entire order can be configured with the `[discount ...]` tag. Discounts are applied on a per−user basis −− you can gate the discount based on membership in a club or other arbitrary means. See *Product Discounts*.

For example, if you decided to adjust the price of T−shirt part number 99−102 up 1.00 when the size is extra large and down 1.00 when the size is small, you would have the following directives defined in <catalog.cfg>:

```
Database          pricing pricing.asc 1
UseModifier       size
PriceAdjustment   size
```

To enable automatic modifier handling, you define a size field in products.txt:

```
code    description   price    size
99−102  T−Shirt       10.00    S=Small, M=Medium, L=Large*, XL=Extra Large
```

You would place the proper tag within your `[item−list]` on the shopping−basket or order page:

```
   [item−accessories size]
```

In the pricing.asc database source, you would need:

```
code      S       XL
99−102    −1.00   1.00
```

If you want to assign a price based on the option, precede the number with an equals sign:

```
code    S       M       L       XL
99−102  =9.00   =10     =10     =11
```

IMPORTANT NOTE: Price adjustments occur AFTER quantity price breaks, so the above would negate anything set with the *PriceBreaks* directive/option.

Numbers that begin with an equals sign (=) are used as absolute prices and are *interpolated for Interchange tags first*, so you can use subroutines to set the price. To facilite coordination with the subroutine, the session variables `item_code` and `item_quantity` are set to the code and quantity of the item being evaluated. They would be accessed in a global subroutine with `$Vend::Session->{item_code}` and `$Vend::Session->{item_quantity}`.

The pricing information must always come from a database because of security.

2.5. Item Attributes

Interchange allows item attributes to be set for each ordered item. This allows a size, color, or other modifier to be attached to a common part number. If multiple attributes are set, then they should be separated by commas. Previous attribute values can be saved by means of a hidden field on a form, and multiple attributes for each item can be *stacked* on top of each other.

The configuration file directive *UseModifier* is used to set the name of the modifier or modifiers. For example

```
UseModifier        size,color
```

will attach both a size and color attribute to each item code that is ordered.

IMPORTANT NOTE: You may not use the following names for attributes:

```
item  group  quantity  code  mv_ib  mv_mi  mv_si
```

You can also set it in scratch with the mv_UseModifier scratch variable -- `[set mv_UseModifier]size color[/set]` has the same effect as above. This allows multiple options to be set for products. Whichever one is in effect at order time will be used. Be careful, you cannot set it more than once on the same page. Setting the `mv_separate_items` or global directive *SeparateItems* places each ordered item on a separate line, simplifying attribute handling. The scratch setting for `mv_separate_items` has the same effect.

The modifier value is accessed in the `[item-list]` loop with the `[item-modifier attribute]` tag, and form input fields are placed with the `[modifier-name attribute]` tag. This is similar to the way that quantity is handled, except that attributes can be "stacked" by setting multiple values in an input form.

You cannot define a modifier name of *code* or *quantity*, as they are already used. You must be sure that no fields in your forms have digits appended to their names if the variable is the same name as the attribute name you select, as the `[modifier-name size]` variables will be placed in the user session as the form variables size0, size1, size2, etc.

You can use the `[loop arg="item item item"]` list to reference multiple display or selection fields for modifiers, or you can use the built-in `[PREFIX-accessories ...]` tags available in most Interchange list operations. The modifier value can then be used to select data from an arbitrary database for attribute selection and display.

Below is a fragment from a shopping basket display form which shows a selectable size with "sticky" setting. Note that this would always be contained within the `[item_list]` `[/item-list]` pair.

```
<SELECT NAME="[modifier-name size]">
<OPTION  [selected [modifier-name size] S]> S
<OPTION  [selected [modifier-name size] M]> M
<OPTION  [selected [modifier-name size] L]> L
<OPTION [selected [modifier-name size] XL]> XL
</SELECT>
```

It could just as easily be done with a radio button group combined with the `[checked ...]` tag.

Interchange will automatically generate the above select box when the `[accessories <code` size]> or `[item-accessories size]` tags are called. They have the syntax:

```
[item_accessories attribute*, type*, field*, database*, name*, outboard*]

[accessories code attribute*, type*, field*, database*, name*, outboard*]
```

code

Not needed for item-accessories, this is the product code of the item to reference.

attribute

The item attribute as specified in the UseModifier configuration directive. Typical are `size` or `color`.

type

The action to be taken. One of:

```
select          Builds a dropdown <SELECT> menu for the attribute.
                NOTE: This is the default.

multiple        Builds a multiple dropdown <SELECT> menu for the
                attribute.  The size is equal to the number of
                option choices.

display         Shows the label text for *only the selected option*.

show            Shows the option choices (no labels) for the option.

radio           Builds a radio box group for the item, with spaces
                separating the elements.

radio nbsp      Builds a radio box group for the item, with  
                separating the elements.

radio left n    Builds a radio box group for the item, inside a
                table, with the checkbox on the left side. If "n"
                is present and is a digit from 2 to 9, it will align
                the options in that many columns.

radio right n   Builds a radio box group for the item, inside a
                table, with the checkbox on the right side. If "n"
                is present and is a digit from 2 to 9, it will align
                the options in that many columns.

check           Builds a checkbox group for the item, with spaces
                separating the elements.
```

```
check nbsp      Builds a checkbox group for the item, with  
                separating the elements.

check left n    Builds a checkbox group for the item, inside a
                table, with the checkbox on the left side. If "n"
                is present and is a digit from 2 to 9, it will align
                the options in that many columns.

check right n   Builds a checkbox group for the item, inside a
                table, with the checkbox on the right side. If "n"
                is present and is a digit from 2 to 9, it will align
                the options in that many columns.
```

The default is 'select', which builds an HTML select form entry for the attribute. Also recognized is 'multiple', which generates a multiple−selection drop down list, 'show', which shows the list of possible attributes, and 'display', which shows the label text for the selected option only.

field

The database field name to be used to build the entry (usually a field in the products database). Defaults to a field named the same as the attribute.

database

The database to find **field** in, defaults to the first products file where the item code is found.

name

Name of the form variable to use if a form is being built. Defaults to mv_order_**attribute** −− i.e. if the attribute is **size**, the form variable will be named **mv_order_size**.

outboard

If calling the item−accessories tag, and you wish to select from an outboard database, you can pass the key to use to find the accessory data.

When called with an attribute, the database is consulted and looks for a comma−separated list of attribute options. They take the form:

```
name=Label Text, name=Label Text*
```

The label text is optional −− if none is given, the **name** will be used.

If an asterisk is the last character of the label text, the item is the default selection. If no default is specified, the first will be the default. An example:

```
[item_accessories color]
```

This will search the product database for a field named "color". If an entry "beige=Almond, gold=Harvest Gold, White*, green=Avocado" is found, a select box like this will be built:

```
<SELECT NAME="mv_order_color">
<OPTION VALUE="beige">Almond
<OPTION VALUE="gold">Harvest Gold
```

```
<OPTION SELECTED>White
<OPTION VALUE="green">Avocado
</SELECT>
```

In combination with the `mv_order_item` and `mv_order_quantity` variables this can be used to allow entry of an attribute at time of order.

If used in an item list, and the user has changed the value, the generated select box will automatically retain the current value the user has selected.

The value can then be displayed with `[item-modifier size]` on the order report, order receipt, or any other page containing an `[item-list]`.

2.6. Product Discounts

Product discounts can be set upon display of any page. The discounts apply only to the customer receiving them, and are of one of three types:

1. A discount for one particular item code (key is the item-code)
2. A discount applying to all item codes (key is ALL_ITEMS)
3. A discount for an individual line item (set the mv_disount attribute with embedded Perl)
4. A discount applied after all items are totaled (key is ENTIRE_ORDER)

The discounts are specified via a formula. The formula is scanned for the variables $q and $s, which are substituted for with the item *quantity* and *subtotal* respectively. The variable $s is saved between iterations, so the discounts are cumulative. In the case of the item and all items discount, the formula must evaluate to a new subtotal for all items *of that code* that are ordered. The discount for the entire order is applied to the entire order, and would normally be a monetary amount to subtract or a flat percentage discount.

Discounts are applied to the effective price of the product, including any quantity discounts or price adjustments.

To apply a straight 20% discount to all items:

```
[discount ALL_ITEMS] $s * .8 [/discount]
```

or with named attributes:

```
[discount code=ALL_ITEMS] $s * .8 [/discount]
```

To take 25% off of only item 00-342:

```
[discount 00-342] $s * .75 [/discount]
```

To subtract $5.00 from the customer's order:

```
[discount ENTIRE_ORDER] $s - 5 [/discount]
```

To reset a discount, set it to the empty string:

```
[discount ALL_ITEMS][/discount]
```

Perl code can be used to apply the discounts, and variables are saved between items and are shared with the `[calc]` tag. This example gives 10% off if two items are ordered, with 5% more for each additional up to a maximum of 30% discount:

```
[calc]
    [item-list]
        $totalq{"[item-code]"} += [item-quantity];
    [/item-list]
        return '';
[/calc]

[item-list]
    [discount code="[item-code]"]
        return ($s)       if $totalq{"[item-code]"} == 1;
        return ($s * .70) if $totalq{"[item-code]"} > 6;
        return ($s * ( 1 - 0.05 * $totalq{"[item-code]"} ));
    [/discount]
[/item-list]
```

Here is an example of a special discount for item code 00-343 which prices the *second* one ordered at 1 cent:

```
[discount 00-343]
return $s if $q == 1;
my $p = $s/$q;
my $t = ($q - 1) * $p;
$t .= 0.01;
return $t;
[/discount]
```

If you want to display the discount amount, use the `[item-discount]` tag.

```
[item-list]
Discount for [item-code]: [item-discount]
[/item-list]
```

Finally, if you want to display the discounted subtotal, you need to use the `[calc]` capability:

```
[item-list]
Discounted subtotal for [item-code]: [currency][calc]
                                        [item-price] * [item-quantity]
                                        [/calc][/currency]
[/item-list]
```

2.7. Sales Tax

Interchange allows calculation of sales tax on a straight percentage basis, with certain items allowed to be tax-exempt. To enable this feature, the directive *SalesTax* is initialized with the name of a field (or fields) on the order form. Commonly, this is zipcode and/or state:

```
SalesTax    zip,state
```

This being done, Interchange assumes the presence of a file `salestax.asc`, which contains a database with the percentages. Each line of `salestax.asc` should be a code (again, usually a five-digit zip or a two letter state) followed by a tab, then a percentage. Example:

```
45056   .0525
```

```
61821   .0725
61801   .075
IL      .0625
OH      .0525
VAT     .15
WA      .08
```

Based on the user's entry of information in the order form, Interchange will look up (for our example SalesTax directive) first the zip, then the state, and apply the percentage to the SUBTOTAL of the order. The subtotal will include any taxable items, and will also include the shipping cost if the state/zip is included in the *TaxShipping* directive. It will add the percentage, then make that available with the `[salestax]` tag for display on the order form. If no match is found, the entry 'default' is applied −− that is normally 0, but can be anything.

If business is being done on a national basis, it is now common to have to collect sales tax for multiple states. If you are doing so, it is possible to subscribe to a service which issues regular updates of the sales tax percentages −− usually by quarterly or monthly subscription. Such a database should be easily converted to Interchange format −− but some systems are rather convoluted, and it will be well to check and see if the program can export to a flat ASCII file format based on zip code.

If some items are not taxable, then you must set up a field in your database which indicates that. You then place the **name** of that field in the *NonTaxableField* directive. If the field for that item evaluates true on a yes−no basis (i.e. is set to `yes`, `y`, 1, or the like), sales tax will not be applied to the item. If it evaluates false, it will be taxed.

If your state taxes shipping, use the *TaxShipping* directive. Utah and Nevada are known to tax shipping −− there may be others.

If you want to set a fixed tax for all orders, as might occur for VAT in some countries, just set the *SalesTax* directive to a value like `tax_code`, and define a variable in the user session to reflect the proper entry in the `salestax.asc` file. To set it to 15% with the above `salestax.asc` file, you would put in a form:

```
<INPUT TYPE=hidden NAME=tax_code VALUE="VAT">
```

or to do it without submitting a form:

```
[perl] $Values->{tax_code} = 'VAT'; return; [/perl]
```

3. THE CHECKOUT PROCESS

3.1. Advanced Multi−level Order Pages

An unlimited number of order checking profiles can be defined with the *OrderProfile* directive, or by defining order profiles in scratch variables. This allows a multi−level ordering process, with checking for format and validity at every stage.

To custom−configure the error message, place it after the format check requirement.

Specifications take the form of an order page variable (like name or address), followed by an equals sign and one of five check types:

required

A non−blank value is required

mandatory

Must be non−blank, and must have been specified on this form, not a saved value from a previous form

phone

The field must look like a phone number, by a very loose specification allowing numbers from all countries

phone_us

Must have US phone number formatting, with area code

state

Must be a US state, including DC and Puerto Rico.

province

Must be a Canadian province or pre−1997 territory.

state_province

Must be a US state or Canadian province.

zip

Must have US postal code formatting, with optional ZIP+4. Also called by the alias `us_postcode`.

ca_postcode

Must have Canadian postal code formatting. Checks for a valid first letter.

postcode

Must have Canadian or US postal code formatting.

true

Field begins with **y**, **1**, or **t** (Yes, 1, or True) − not case sensitive

false

Field begins with **n**, **0**, or **f** (No, 0, or False) − not case sensitive

email

Rudimentary email address check, must have an '@' sign, a name, and a minimal domain

regex

A regular expression to check against. To check that all submissions of the "foo" variable have "bar" at the beginning, do:

```
foo=regex ^bar
```

You can add an error message by putting it in quotes at the end:

```
foo=regex ^bar "You must have bar at the beginning of this"
```

If you want to use a backslash to introduce a Perl literal like `\w`, you must double the backslash, i.e.

```
foo=regex ^bar\\w+$ "You must have 'bar' followed by only word characters"
```

length

A range of lengths you want the input to be:

```
foo=length 4-10
```

That will require `foo` be from 4 to 10 characters long.

unique

Tests to see that the value would be a unique key in a table:

```
foo=unique userdb Sorry, that username is already taken
```

filter

Runs the value through an Interchange filter and checks that the returned value is equal to the original value.

```
foo=filter entities Sorry, no HTML allowed
```

To check for all lower−case characters:

```
foo=filter lower Sorry, no uppercase characters
```

Also, there are pragmas that can be used to change behavior:

&charge

Perform a real−time charge operation. If set to any value but "custom", it will use Interchange's CyberCash routines. To set to something else, use the value "custom ROUTINE". The ROUTINE should be a GlobalSub which will cause the charge operation to occur −− if it returns non−blank, non−zero the profile will have succeeded. If it returns 0 or undef or blank, the profile will return failure.

&credit_card

Checks the mv_credit_card_* variables for validity. If set to "standard", it will use Interchange's `encrypt_standard_cc` routines. This destroys the CGI value of mv_credit_card_number −− if you don't want that to happen (perhaps to save it for sending to CyberCash) then add the word `keep` on the end. Example:

```
# Checks credit card number and destroys number after encryption
# The charge operation can never work

&credit_card=standard
&charge=custom authorizenet

# Checks credit card number and keeps number after encryption
# The charge operation can now work

&credit_card=standard keep
&charge=custom authorizenet
```

You can supply your own check routine with a GlobalSub:

```
&credit_card=check_cc
```

The `GlobalSub` check_cc will be used to check and encrypt the credit card number, and its return value will be used to determine profile success.

&fail

Sets the mv_failpage value.

```
&fail=page4
```

If the submit process succeeds, the user will be sent to the page `page4`.

&fatal

Set to '&fatal=yes' if an error should generate the error page.

&final

Set to '&final=yes' if a successful check should cause the order to be placed.

&return

Causes profile processing to terminate with either a success or failure depending on what follows. If it is non−blank and non−zero, the profile succeeds.

```
# Success :)
&return 1

# Failure :\
&return 0
```

Will ignore the &fatal pragma, but &final is still in effect if set.

&set

Set a user session variable to a value, i.e. `&set=mv_email [value email]`. This will not cause failure if blank or zero.

&setcheck

Set a user session variable to a value, i.e. `&set=mv_email [value email]`. This **will** cause failure if set to a blank or zero. It is usually placed at the end after a &fatal pragma would have caused the process to stop if there was an error −− can also be used to determine pass/fail based on a derived value, as it will cause failure if it evaluates to zero or a blank value.

&success

Sets the mv_successpage value. Example:

```
&success=page5
```

If the submit process succeeds, the user will be sent to the page `page5`.

As an added measure of control, the specification is evaluated for the special Interchange tags to provide conditional setting of order parameters. With the `[perl] [/perl]` capability, quite complex checks can be done. Also, the name of the page to be displayed on an error can be set in the `mv_failpage` variable.

The following file specifies a simple check of formatted parameters:

```
name=required You must give us your name.
address=required Oops! No address.
city=required
state=required
zip=required
email=required
phone_day=phone_us XXX−XXX−XXXX phone−number for US or Canada
&fatal=yes
email=email Email address missing the domain?
&set=mv_email [value email]
&set=mv_successpage ord/shipping
```

The profile above only performs the &set directives if all of the previous checks have passed −− the &fatal=yes will stop processing after the check of the email address if any of the previous checks failed.

If you want to place multiple order profiles in the same file, separate them with __END__, which must be on a line by itself.

User−defined check routines can be defined in a GlobalSub:

```
GlobalSub <<EOF
sub set_up_extra_check {
    BEGIN {
        package Vend::Order;
        sub _pt_postcode {
            # $ref is to Vend::Session->{'values'} hash
            # $var is the passed name of the variable
            # $val is current value of checked variable
            my($ref, $var, $val) = @_;

            if ($ref->{country} =~ /^(PT|portugal)$/i) {
                return $val =~ /^\d\d\d\d$/ ?
                    (1, $var, '') : (undef, $var, "not a Portugese postal code");
            }
            else {
                return (1, $var, '');
            }
        }
    }
}
EOF
```

Now you can specify in an order profile:

```
postcode=pt_postcode
```

Very elaborate checks are possible. There must be an underscore preceding the routine name. The return value of the subroutine should be a three−element array, consisting of:

1. the pass/fail ('1' or 'undef') status of the check;
2. the name of the variable which was checked;
3. a standard error message for the failure, in case a custom one has not been specified in the order profile.

The latter two elements are used by the `[error]` tag for on−screen display of form errors. The checkout page of the Foundation demo includes examples of this.

3.2. Simple Order Report File

The simple order report file, "report", defines the layout of the order report which gets mailed on the completion of the order. For example,

```
             Order Date: $date

                   Name: $name
          Email address: $email

       Shipping address: $addr
       Town, State, Zip: $town, $state $zip
                Country: $country
```

Any input field from the order page can be included using the dollar sign notation.

To prevent a value from being included in the order report, just add it to the **ReportIgnore** configuration directive.

Interchange defines some values for use in the search form −− they begin with `mv_` and are automatically ignored.

3.3. Fully−configurable Order Reports

You can specify a fully−configurable order report by setting the hidden field "mv_order_report" to a legal Interchange page. This page will be interpolated with all Interchange tags before sending by email. The order number as set by backend ordering is in the variable mv_order_number, and available for your use.

You could if you wish include HTML in the file, which will be interpreted by many mailers, but you can choose to use standard ASCII format. An example report is provided in the demo file <pages/ord/report.html>.

3.4. Order Receipts

The file can of course be configured with all Interchange tags, and will be interpolated for the ordered items before they are deleted from the user session. You can set the default receipt with the `receipt` key in SpecialPage. If using order *Route*s, as in the `foundation` demo, you set it with the `receipt` key to `Route`.

3.5. The Order Counter

An order counter can be enabled if the *OrderCounter* directive is set to a file name. An incrementing count of all orders will be kept and assigned as orders are placed. By default, the number starts at 0, but you can edit the file and change the default starting number at any time.

This capability is made possible by the File::CounterFile module, available (as a part of the fine libwww modules) at the same place you got Interchange. It is included with the distribution.

3.6. Customer Input Fields

On the order (or shopping basket) page, by default order.html, you will have a number of input fields allowing the customer to enter information such as their name and address. You can add more fields simply by putting more input elements on the order.html page, and the information will automatically be included in the order report. Input elements should be written in this way:

```
<input type="text" name="town" value="[value town]" size=30>
```

Choose a name for this input field such as "email" for an email address. Set the name attribute to the name you have chosen.

The value attribute specifies the default value to give the field when the page is displayed. Because the customer may enter information on the order page, return to browsing, and come back to the order page, you want the default value to be what was entered the first time. This is done with the `[value var]` element,

which returns the last value of an input field. Thus,

```
value="[value name]"
```

will evaluate to the name entered on the previous order screen, such as:

```
value="Jane Smith"
```

which will be displayed by the browser.

The size attributes specifies how many characters wide the input field should be on the browser. You do not need to set this to fit the longest possible value since the browser will scroll the field, but you should set it large enough to be comfortable for the customer.
