

# The Demystification of Lasso Custom Types

## Using Custom Types to Improve Code Readability, Reliability, and Maintainability

Greg Willits • Feb 18, 2005 • Updated Oct 1, 2006

<http://www.ldml.org/articles/demystifytypes.lasso>

You've probably heard of Lasso's custom types and thought they were for advanced developers doing tricky things that you'd never be interested in. In this article, my goal is to dissolve your fears, and to help you see how simple custom types are, and how to use them to improve the readability, reliability, and maintainability of your code.

It's true that custom types can do tricky things, and that custom types are the basis for doing complex object-oriented programming. However, it is also true that custom types do not require that you be doing tricky things to take advantage of them, and you do not have to be an OOP guru to understand them.

Yes, by the time we're done, I'm going to throw some big words at you, but you're going to be pleasantly surprised at the simple concepts these über-geek words really represent. While we will talk about some object-oriented concepts, I will not attempt to convert you to doing everything with OOP (I don't do everything with OOP myself), but rather I just want to help demystify the terms, and show you how they apply to practical every-day programming.

### What is a Custom Type?

A custom data type is nothing more than regular Lasso code structured in such a way that you can store multiple data chunks in a single variable, and perform custom actions on those data chunks using the Lasso member tag syntax.

A standard lasso variable can hold data (even multiple chunks of data using arrays and maps), but there's only a few preset actions that can be performed on that data. For example, if we had an array of purchase order line item SKUs, there is, of course, no member tag to check the in-stock status of each item in that array. To do that, we have to write some custom code that iterates through that array and does whatever is necessary such as look for each item in an inventory database table. So, arrays are great, but they're very simple.

A custom tag can perform specialized actions, but it cannot hold data. We could write a custom tag to do the database searches mentioned above, but the tag cannot store the results. It executes its code, then everything disappears. We'd have to create another array to store the results of each SKU search, or reformat our original array into an array of pairs or maps. In this example, we might have up to three separate parts to our solution: the original array, a custom tag to search for stock status of a SKU, and a results array.

A custom type allows us to combine all these pieces into a single entity which uses the same `$variable->memberTag` syntax of native lasso data types (strings, arrays, maps, etc). Just like we might have a string variable on which we find the size using `$x->size`, or remove a prefix using `$x->(removeLeading:'pre-')`, we can use custom types to have something that ends up looking like `$myPurchaseOrder->(getStatus:-sku='xyz')` to perform custom actions and something like `$myPurchaseOrder->'shipDate'` to retrieve a specific piece of data. Ultimately, how we code with a custom type is just like any other Lasso variable and member tag combination.

## The Path to CTypes is CTags

I need to make sure you understand what a custom tag is and how they work before we dig into custom type coding specifics. Custom types are comprised of a little overhead code, then a bunch of custom tag code; so, you really need to understand custom tags first. Of course, read the Lasso documentation on custom tags, but here's an overview.

A custom tag takes a chunk of Lasso code, and puts it in a special wrapper. The wrapper is what makes a custom tag useful. That wrapper gives the code a name which is then used like any Lasso tag name, allows us to pass parameters if needed just like a Lasso tag, and it allows us to retrieve results.

Let's say you want to calculate a person's age based on the birthday and today's date. The reliable way to do that in Lasso looks like this:

```
var:'age' = math_floor:  
  ($birthdate->(Difference:(date), -second))  
  / (3600*24*365.25);
```

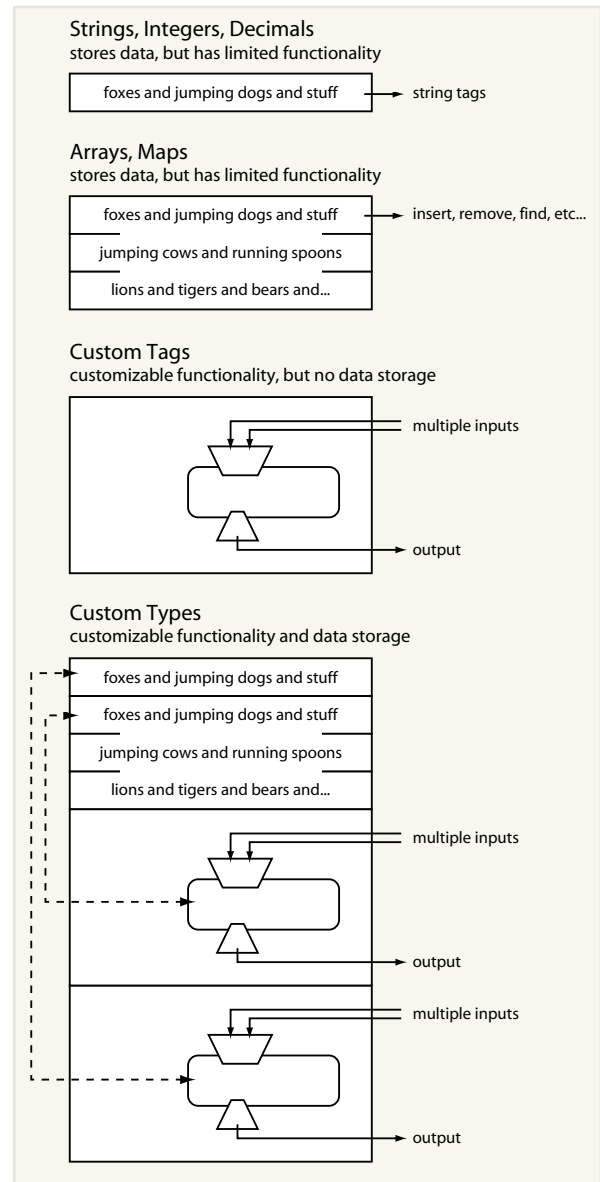
If you needed to do that ten times all over your web site, it'd be a bit painful to write out every time, and would be prone to errors every time you wrote it. It would be much better if we could turn that into a standard function and use it just like a lasso tag. We'd have one parameter to pass, the birthdate, and we want one result back, the age in years. If this were a Lasso tag, our application code could look something like this:

```
var:'age' = calculateAge:$someBirthdate;
```

Much cleaner to read. Much easier to write. Much less likely to create a bug each time we use it. Indeed, creating tags like that is what a custom tag does. Here's what the custom tag code would look like.

```
define_tag:'calculateAge',  
  -required = 'birthdate';  
  
  return: math_floor:(#birthdate->  
    (difference:(date), -second))/  
    (3600*24*365.25);  
  
/define_tag;
```

We use the `define_tag` container tag (that wrapper I mentioned), which names the tag, and defines the expected inputs. The return tag is



used to pass the result back out of the tag. Tags can be used to house as complicated a chunk of code as it needs to be. Most tags shouldn't be more than several dozen lines long to keep their functionality focused, but I have a few custom tags that are hundreds of lines long. Anything you have ever written in an include so you could use it more than once, could, in theory, be a custom tag.

Just to provide a more realistic picture of a custom tag, let me make a more complex example of the age calculation tag. Maybe we don't always want the calculation based on today's date. We may need the age of a student as of September 1st when school started. So we need an `AsOf` date. That code could look like this:

```

define_tag:'calculateAge',
  -priority='replace',
  -required='birthdate',
  -optional='asOfDate';

if: !local_defined:'asOfDate';
  local:'asOfDate' = null;
/if;

#birthdate = date:#birthdate;
#asOfDate = date:#asOfDate;

if: !((#asOfDate->type == 'date')
  && (#asOfDate->year > 0));
  #asOfDate = date;
/if;

if: (#birthdate->type == 'date')
  && (#birthdate->year > 0);
  return: math_floor: (#birthdate->
    (difference:#asOfDate, -second))/
    (3600*24*365.25);
/if;

/define_tag;

```

I will leave it up to you to work with the Lasso documentation and your Lasso buddies to figure out the details if you need to. What you need to take away at this point is a recognition of how Lasso code can be written inside a custom tag wrapper so it can be used just like a regular lasso tag with parameters and results. If you understand that concept, even if you don't understand the syntax details, then we're good to move on.

## Custom Type Syntax Structure

I want to start the custom type discussion by first introducing the structure of a custom type. A custom type is made up of a few distinct pieces. Together they may look daunting, but when broken apart, you should see that it is really not complicated. Most likely you're already used to writing LDML code with several inlines to perform multiple database operations. In such code you'd have:

```

inline:
  // do some stuff here
/inline;

inline:
  // do some more stuff here
/inline;

```

At a high level, the custom type code isn't a whole lot different than that. A custom type is not much more complicated than a set of custom tags collected together between a `define_type` wrapper. A typical custom type looks like this:

```

define_type:'purchaseOrder';

// In this section, called the initializer,
// you will define the properties or
// attributes of your data.
// That is accomplished by declaring
// local variables.

local:
  'x'   = string,
  'y'   = integer;

define_tag:'onCreate';
  // this is a special purpose tag
  // to initialize your data
  // we'll look at this more later
/define_tag;

define_tag:'getShipDate';
  // this would be a member tag
  // which would perform some operation
  // to determine the ship date of our
  // purchase order data type
/define_tag;

define_tag:'whatever';
  // we can have as many member tags
  // defined as we need
/define_tag;
/define_type;

```

When you look at that structure, I hope you see that a custom type is not all that complicated. It amounts to little more than some local variables and custom tags all between a `define_type` container.

### The CType Initializer

Proper custom type design says that the only thing that should be coded in the initializer is local variable declarations. For backwards compatibility reasons, Lasso doesn't actually enforce that, so you can write any LDML code there you want, and technically it will run. However, Lasso (especially as of LP8) can do some things that makes custom types much faster if the only thing in the initializer is local variable declarations. All other code should be inside a tag. Use the `onCreate` tag to do anything needed to set up the ctype.

## Custom Type Properties and Local Variables

The one tricky part of a custom type's structure, if it can even be called tricky, is the relationship of local variables inside the custom type and all the custom tags. To sort through how local variables work inside custom types, let's first look at the initializer. The initializer is technically any code inside the `define_type` tags, but outside the `define_tag` tags. That's usually written immediately after the `define_type` tag and above the first `define_tag`, but the code could exist between `define_tag` tags (but don't do that). In the sample structure above that would be the local variable declarations.

The initializer is where we declare all variables that need to be accessible to more than one member tag, and/or available to use outside the custom type code (i.e. to retrieve data using the typical method of `$myType->'color'`). If you want a variable inside a custom type to be exposed to your application code to read or to set, then it must be in this initializer section. The term *exposed* means a variable or member tag that is directly accessible to your application code. Let's look at some code, then see what the rules are.

```
define_type: 'purchaseOrder';

local:
  'totalAmount' = decimal,
  'salesTax'    = decimal;

define_tag: 'calcTotalAmount';
  local: 'calcResults' = (decimal);
  // do a bunch of calcs
  (self->'totalAmount') = #calcResults;
/define_tag;

define_tag: 'calcSalesTax';
  (self->'salesTax') =
    (self->'totalAmount') * 0.10;
/define_tag;
/define_type;
```

The above code illustrates these rules:

- to have a variable available as a property of the custom type (Lasso calls them *instance variables*) with the syntax `$myType->'color'`, the variable must be declared as a local in the initializer
- `var totalAmount` is available as a property of the custom type as `$myType->'totalAmount'`
- `var salesTax` is available as a property of the custom type as `$myType->'salesTax'`
- `var calcResults` is not an instance variable. It is *not* available as `$myType->'calcResults'`
- `var calcResults` is available only inside the `->calcResults` tag, and must use the `#calcResults` syntax. It is not even available to use inside the `->calcSalesTax` tag.
- vars `totalAmount` and `salesTax` are available to all members tags in the ctype, and must be used with the `self->'xyz'` syntax

If you think about it, all these rules can be summed up as `#vars` are truly local to only the member tag in which they are defined. Locals in the initializer are scoped to the whole custom type and can be used by any member tag using the `self->` tag to accomplish that, and they are available to be used as properties of the ctype.

Back to that whole *exposed* thing — we would say that the variables `totalAmount` and `salesTax` are exposed to the application. The local `var calcResults` is not.

## Custom Type Member Tags and Encapsulation

We've looked at custom type properties, so now let's look at custom type member tags (*methods* in most other languages). Generally, there is very little that is different between a regular custom tag, and a member tag. If you're quite comfortable with ctags, then you're well on your way to being a custom type guru too. That being said, there's a couple things to discuss.

One question that inevitably comes up about custom tags in general, and therefore in custom types as well, is whether a custom tag (member tag) can use `$var` style page variables. The technical answer is yes. However, there's many reasons why that usually should not be done. As you get more sophisticated in your use of custom types, you'll come to understand that a major part of their beauty is rooted in a concept called *encapsulation*. Using page variables directly instead of passing them as inputs and outputs destroys the integrity of encapsulation. Without that integrity, we lose the reliability and maintainability that are two of our goals in using custom types to start with.

Considering the goals of custom types and their value if used “correctly,” member tags should never be programmed to use page variables in the member tag code which are also manipulated by processes outside the custom type code.

The one area where using page variables inside custom types is justifiable is when many ctypes are developed to work together and they all share common environment configuration settings. For example, if you have several custom types which generate database queries, then for each inline block you have to pass the Lasso username and password. Should those be passed as inputs to each member tag that has a query? That’s quite a waste of time really. If the name and password are defined with standard variables for the whole application, or better yet, for a generic framework common to many applications, then certainly using those page variables inside the member tag code is justifiable. The whole point of having environment configuration settings and even frameworks is to save you from repetitively coding the same inputs. So, we would go ahead and extend that inside member tags as well, but it must be to the overall advantage of the goals of the software, and not some lazy way out of doing things in a more proper, structured approach. This is a case of having a very good reason to go against the rule.

However, if your custom type code has to work outside this standardized environment, then it must allow for parameter inputs to substitute for those page vars. It would be bad design to say, “here’s my ctype, it must have page vars \$x and \$y defined to work.” It is better to preserve encapsulation, and pass the equivalent of \$x and \$y as parameters. Of course, another option is to make all the related ctypes available to the developer including the environment config setup, and code everything so that the var names of the config are all coherently grouped (i.e. all vars have a common prefix or something). At any rate, the point is that custom types are advantageous in part because of encapsulation, and that advantage should be preserved in the design and implementation of the ctype. The less you know about how and where the ctype will be used, the more strict the encapsulation should be adhered to.

## Big Word #1: Encapsulation

This is an Object Oriented Programming term. It means that data and code which manipulates that data are contained within a single programming unit. In Lasso, that single unit is the custom type. The intent is that the data in the ctype cannot be acted upon except through the code that is in the ctype. If that is the case, then anything that causes errors in that encapsulated data must be caused by the encapsulated code. This keeps code which alters the data on a very short leash. It’s easier to track and debug; and, once it is debugged, it should be perfectly reliable every time it is implemented.

## onCreate—A Special Member Tag

The onCreate member tag shown in my code examples is a special case. This tag is never called by the developer. Lasso will automatically call this tag when a variable is created and set to the custom data type. So, when your code processes a statement like `var: 'x' = purchaseOrder;` Lasso will automatically run the onCreate tag at that point. Therefore, we use this tag to perform any setup that is needed before we can call any of the member tags. Often, this means passing initialization parameters in the var declaration.

It’s important to note here that unlike `define_tag`, `define_type` does not accept parameters in the same way. Prior to Lasso 8, there is no equivalent to the `-required` and `-optional` commands that `define_tag` has. If your code looks like this:

```
var: 'x' = (myCustomType:
  -paramA = $someVar,
  -paramB = true);
```

then the onCreate tag is where you process the parameters and do whatever is needed by reading from the params tag.

If you are using Lasso 8 or newer, then things are fairly easy as the onCreate tag does allow the `-required` and `-optional` parameters, and onCreate will automatically receive parameters like a normal custom tag, but you do have to use self to pass them into the initializer locals like this:

```

define_type: 'anLP8CustomType';

local:
  'paramA' = (string),
  'paramB' = (boolean),
  'error'  = (string);

define_tag: 'onCreate',
  -required = 'paramA',
  -optional = 'paramA';

  self->'paramA' = #paramA;
  self->'paramB' = local:'paramB';

```

However, if you're using LP 5 through 7, then things are a little trickier.

```

define_type: 'anLP7CustomType';

local:
  'paramA' = (string),
  'paramB' = (boolean),
  'error'  = (string);

define_tag: 'onCreate';
  if: (params->(find:-paramA'))->size > 0;
    (self->'paramA') =
      (params->(find:-paramA'))->
        (get:1)->second;
  else;
    // do some error control
  /if;

  if: (params->(find:-paramB'))->size > 0;
    (self->'paramB') =
      (params->(find:-paramB'))->
        (get:1)->second;
  else;
    // do some error control
  /if;
/define_tag;

```

There's many ways that error handling can be written, so don't focus on that, but know that some form of input checking should be done because Lasso isn't going to handle that for you like it can in custom tags. Also, there's actually several ways to code the pulling of param data from the `params` array to populate the instance variables. In this case, I'm trying to be rather obvious about what's happening.

Note: in LP5 and 6, the `params` array is not available to the `onCreate` tag. You have to set a local equal to `params` (essentially copying them), then inside the `onCreate` tag refer to that local. Like this:

```

define_type: 'anLP5or6CustomType';

local:
  'myParams' = params,
  etc...

define_tag: 'onCreate';
// thanks to James Harvard for this trick
if: (self->'myParams')->
  (find:-paramA'))->size > 0;

  (self->'paramA') =
    ((self->'myParams')->
      (find:-paramA'))->(get:1)->second;
else;
  // do some error control
/if;

```

One more point about `onCreate` to make.

Unlike other custom tags within a custom type, `onCreate` cannot use the return tag. `onCreate` is for initializing properties of the custom type. It's not really supposed to do any "work" nor return any results. Once a var has been declared as your custom type, you can follow that up with checking the values of instance variables. So, we might code using the above sample `myCustomType` like this:

```

var: 'x' = (myCustomType:
  -paramA = $someVar,
  -paramB = true);

  if: ($x->'error');
    // oh man!
  else;
    // we're good
  /if;

```

That covers the most important points about the main syntax of writing custom types. Of course, the Lasso documentation covers many more details and advanced capabilities, but these are the essentials.

## Why Use Custom Types

In the sections above, we've looked at the structure of a custom type and some of the basics regarding the syntax and programming of instance variables (a.k.a. properties) and member tags (a.k.a. methods). So now the big question, why and/or when should custom types be used? We considered one attribute of custom types that gave us two reasons why: encapsulation promotes efficient maintenance (easier debugging) and reliability (self-contained reusable code is hard to break once it has proven to work in all intended scenarios).

Before we delve into more justifications, I think we need to look at more detailed example of a custom type in action. Let us consider a purchase order. This is a complex data structure with any number of actions that may need to be performed on it, or that as a component of an even more complex process, it may need to provide data for. I'm not going to go hog wild here, just a little building on the types of samples I've provided earlier. First, let's look at the custom type code (assuming LP8):

```
define_type: 'purchaseOrder';

// our data type's properties
local:
  'poNumber'      = (string),
  'error'         = (integer),
  'linesOutOfStock' = (integer),
  'shipDate'     = (date),
  'invoiceDue'   = (boolean),
  'invoiceNo'    = (string),
  'invoiceAmt'   = (decimal),
  'invoiceDate'  = (date);

//-----
define_tag: 'onCreate',
  -required = 'poNumber';
  self->'poNumber' = #poNumber;
/define_tag;

//-----
define_tag: 'getStockStatus';
  // returns number of line items not in stock

  local: 'sqlActn' = (string);
  self->'error' = 0;

  // execute some crazy SQL here using the poNumber
  #sqlActn += 'SELECT stockStatus FROM .... ';
  #sqlActn += ' WHERE .... =\'' + encode_sql:(self->'poNumber') + '\'. ....';
  #sqlActn += ' AND stockStatus=\''Out of Stock\'';

  // we use standard environment config $vars here
  inline:
    -username = $queryUser,
    -password = $queryPswd,
    -database = $db_purchaseOrders,
    -sql = #sqlActn;

  self->'error' = error_code;
  self->'linesOutOfStock' = found_count;

/inline;
/define_tag;
```

```

//-----
define_tag:'getShipStatus';
    // populates the instances var for
    // the scheduled ship date
/define_tag;

//-----
define_tag:'getPaidStatus';
    // populates the instances vars for
    // invoiceDue, invoiceNo, invoiceAmt, invoiceDate
/define_tag;

/define_type;

```

In this custom type, we have a number of properties of a purchase order and a few actions that are typically performed. The code has our typical structure with instance vars declared in the initializer area, an onCreate tag to capture input parameters, and some member tags for custom processes that will manipulate the instance vars. I've tried not to be too detailed, as the code itself doesn't matter as much as the concepts of what the code is supposed to do. The point being that each custom tag can represent some extensive complex code (or even some very simple code). In case you're wondering – with many tags, this can indeed make for some long files (I think my longest custom type is about 3,000 lines, but that's an insane one). In another article, I'll discuss more about good custom type design which will cover issues like how long is too long, and when should a custom type be broken down into more than one.

In our application logic code for a page using this ctype, we might have something to this effect:

```

var:'thisPO' = (ctype_purchaseOrder:
                $myPONumber);

$thisPO->getStockStatus;

if: ($thisPO->'linesOutOfStock') == 0;
    $thisPO->getShipStatus;
/if;

if: ($thisPO->'shipDate') < date;
    $thisPO->getPaidStatus;
/if;

```

Now, right away, I have to point out the obvious. Wasn't that really easy to read? The purpose of the page this code executes in is to check the status of an order and go through some optional fact finding. We check stock. If everything is in stock, we look up the ship status. If the stuff has shipped, we fetch the invoice

status. Whether or not that workflow makes sense in the real world, we just did a whole lot of work in very few lines of code. All the complex stuff is tucked away in our tidy, encapsulated custom type. As we read this code, we don't have to wade through tons of details that really don't matter to the overall purpose at hand. We can focus on the big picture and get the drift of the work flow very quickly.

If we used only custom tags and not custom types, we'd have to create and manage several variables. Not painful to do once, but more work than necessary if this process is something we have to do on multiple pages. If we just used an include to reduce the clutter, there's too many variables and lines of code to be sure that if we have to reuse the code for another page that something won't interfere and break it. With the ctype, everything is nicely contained.

Of course, not every programming challenge is best solved by ctypes, and we'll talk more about that later, but I hope you can derive the main points being made here without focusing too much on the efficacy of any one example.

Seeing these multiple steps which really just populate the properties, you might ask if we couldn't just put all those tags into the onCreate tag so our vars are automatically populated when the \$thisPO variable is created. Technically, yes we could, and in some cases, that may be prudent. In this case, I'll offer that it is likely not a good idea. The main reason is processing expense and task resolution. Some of those member tags may be very process intensive. Based on our workflow, there's no point in looking up the invoice if the order hasn't even shipped yet. So, there's no point in wasting the processing cycles and RAM in performing that work as a default exercise in onCreate. The waste would be magnified if 40% of the calls to this page result in finding orders not yet shipped. On the flip side, there will be cases where the processing task is small, and the

majority of time, the data will be needed, so there can be justification for doing more up front work in onCreate.

Let's now look at our display code where we could have something like this:

```
[if: $thisPO->'linesOutOfStock']
<p>Sorry, your order still has [$thisPO->
'linesOutOfStock'] line items not yet in
stock.</p>
[else]
<p>All items are in stock, and you order is
scheduled to ship on [$thisPO->'shipDate'].
</p>
[/if]

[if: $thisPO->'invoiceDue']
  <p>Our invoice [$thisPO->'invoiceNo'] for
the amount of [$thisPO->'invoiceAmt'] is due
[$thisPO->'invoiceDate']</p>
[/if]
```

As you can see, between the logic and display code, there was no need to set variables because all the variables were already created inside the custom type. Looking at the display code, you might be thinking it is not all that different from using regular variables. But, in a way, that's the point—it really isn't very different to use a custom type. Compared to simple variables, there's a little more typing, but compared to using maps, there is actually less typing in not having to use the `->find` tag. Also, the `object->variable` relationship and syntax improves code readability by helping us see exactly where and how the data was created. If we have a simple `$lineOutOfStock` variable, we really have no idea where that variable was created, or what process modified it. By encapsulating the variable inside a custom type, we know exactly where to look to read about the process that created the data in the variable. Not only is this more readable, helping us to understand the code, but it helps maintainability in making us more efficient in tracking down code which needs to be updated.

## Enhancing Readability Even More

So far I've suggested that OOP code can be more readable than standard procedural code because the bulk of the details are tucked away so that the major steps and work flow of the code stands out as an outline.

We can improve upon this advantage by carefully crafting the names of the instance variables and member tags. Instance variables are nouns, and member tags are used to perform actions, so their names should reflect that by using verbs. Likewise a variable which is a custom type represents either a tangible or conceptual object, and its name should clearly reflect what that is. When you put these names together, you should be reading code that sounds a lot like a regular sentence. Consider this short example:

```
if: ($thisPO->'linesOutOfStock') == 0;
    $thisPO->getShipStatus;
/if;
```

It's pretty clear here what is going on, but the nuances of the naming are important. `linesOutOfStock` reads better in context than `outOfStockLines`. When you think about the name of an instance variable, don't use a name that sounds best as a label. Use a name that sounds best when used in context of the code. As for the methods, `getShipStatus` is more clear than simply using `ShipStatus`. The latter sounds like a label for a noun, and the member tag is not a noun, it is an action. Another key factor in naming member tags is to form the name as a command not a question. Use `getShipStatus` not `whatIsShipStatus`. There can be exceptions to that, but philosophically, in the application code, you want to be telling your objects (ctypes) what to do, not asking them what they know.

## The Advantages of Inheritance

Personally, I find encapsulation to be one of the most compelling reasons to use custom types, even for very simple applications and web pages. As the application gets more complex, there are other very useful advantages to custom types which they derive from their object-oriented programming lineage. The other major advantage I have grown more appreciative of is the ability to manipulate the object, tag, and instance variable names to generate that easy readability. Done well, and the code can require very little commenting to explain what is going on. It can be much easier to come back to code you wrote a year ago and quickly understand what it is doing.

The next reason for using custom types is the concept of *inheritance*. One of the goals of using techniques like includes, custom tags, and custom

types is to take repetitive code and make it easier to reuse from a single source rather than have several copies of it throughout the application. Custom tags and custom types go a little farther than includes in actually trying to encourage that we write code so that it can be reusable by using input parameters. Custom types can take this even further with the concept of inheritance.

Because of inheritance we can look at our code and find properties and member tags that might be shared among a number of similar objects. We can write a generic ctype which includes those features, then write more specific ctypes to extend that generic one and only have to write the additional unique code. In doing this, we might have several ctypes which all literally inherit a part of their code from a common source. This improves reliability because that code is not rewritten several times which introduces new bugs. It also improves maintainability because changes to those shared features can be done in one place.

As an example to illustrate inheritance, let's consider some routines that dynamically draw typical on screen form inputs of popup menus, checkbox lists, and radio button lists. At the detail level we have three distinct objects and code required to draw them. However, they each share some common attributes. Each needs to have a source list of items to include in their list, and perhaps need to read that list from disk, or acquire it from a database field. Each also needs to store a currently selected item. However, each one needs different HTML code to draw them. We can take the common features and create a generic custom type named `valueList`. The code for this ctype might look like this:

```
define_type:'valueList';

locals:
  'listSource'      = (string),
  'listChoices'    = (array),
  'listSelection'  = (string);

define_tag:'onCreate';
  // assume listSource is a disk file
  self->'listSource' = params->(get:1);
/define_tag;

define_tag:'getList';
  // a routine to read a disk file
  // and convert the text to an array
  // which is stored to listChoices
/define_tag;
/define_type;
```

## Big Word #2: Inheritance

This is an Object Oriented Programming concept. Includes and custom tags are pretty much fixed. We write them, they do what they do, and that's it. With custom types we can actually use one custom type as the foundation for writing another custom type without duplicating the code. When creating a custom type, we can say that it *extends* another custom type. When we do that, the second ctype automatically inherits all the properties and member tags of the first ctype. If `$y` is based on `$x`, we can use the member tags of `$x` as though they were written into the code for `$y`.

So, this custom type is pretty simple. It holds some properties, and it can really only do one thing which is to read a disk file to retrieve a list of choices and create an array from that. It doesn't draw any `valueList`. To add that we now need to create three separate ctypes. One each for the three types of HTML lists we want.

```
define_type:'popup', 'valueList';
  define_tag:'drawPopup';
    // code here to create HTML
    // for a popup menu
  /define_tag;
/define_type;

define_type:'checklist', 'valueList';
  define_tag:'drawCheckList';
    // code here to create HTML
    // for a checklist
  /define_tag;
/define_type;

define_type:'radiobtns', 'valueList';
  define_tag:'drawRadioBtns';
    // code here to create HTML
    // for radiobtns
  /define_tag;
/define_type;
```

You'll notice some things which are different about these custom types. First, the `define_type` tag has two names instead of just one. The first name is the name of the ctype being coded, and the second name is the name of another Lasso data type (a custom type, or actually even a built-in type like `array` or `map`). This tells the new ctype that it is to inherit all the properties and tags of the second named data type.

You'll also notice that there are no locals defined for the three specific ctypes. That's

because the custom type popup will use the locals defined in the `valueList` ctype code. Same with the member tags, so there is no need for an `onCreate` tag. So, we could create a popup list object by simply doing this:

```
var: 'colors' =  
    (popup: '/configs/colorList.cnfg');
```

and then calling the `getList` tag by coding:

```
$colors->getList;
```

So, another advantage of using custom types is this ability to use inheritance so we can separate out reusable code into a generic ctype, then create sub-types which code only the unique aspects of that specific type. Other languages call these kinds of ctypes like the `valueList` example abstract. They're abstract because they don't do anything on their own. A `valueList` is not a tangible thing. It needs to be inherited by another ctype before its code is useful. This ability to separate reusable code again provides reliability and maintainability by centralizing common code into an encapsulated container that is easier to test and modify.

You might be thinking, why not write one custom type and just put three different `->draw` members tags in it, so you can just draw which ever one you want as needed. That is tempting. It feels like less work: one custom type instead of four. However, it violates an important principle of good object oriented programming which says that each custom type should do one thing and do it well. Now, by saying *one thing* we're not talking about having one member tag, we're talking about a purpose and a focus. In this case we want our one thing to be a popup menu. The object itself is either a popup menu or a checklist. It cannot be both. Therefore, the code should not try to be both. If you have code that tries to be both a popup menu and a checklist, it can be harder to write the code, harder to test it, and harder to modify it later.

That may not seem problematic for simple objects like these value lists, but it could be a much messier problem for something as complex as a purchase order. If we extend the combining thinking, we could say that a purchase order is just a form just like a material requisition, so we may as well code a custom type that combines POs and Reqs because they share several common features. I hope you can see that coding, testing, and updating the code for a such a custom type would be more difficult having to watch out for PO vs Req features vs. two clearly purposed

### Big Word #3: Polymorphism

This is an Object Oriented Programming concept which describes the ability of a piece of code to yield different behaviors from the same commands. In our example of the value lists, the `valueList` ctype could be made polymorphic (it is, except I skipped a few details in the sample code). The `valueList` ctype would have a tag called `->draw` which wouldn't actually do anything except just be there to establish that a draw command is available. Each of the sub-types like popup, would also have a draw member tag, but each one would have specific code to write out the specific HTML needed for that value list type. The ctype `valueList` would then be polymorphic because the same draw command can result in multiple behaviors (drawing a popup vs a series of checkboxes).

custom types. So, whether the custom type is small or large, it is best to develop with the philosophy of "do one thing, and do it well."

### Every Day Uses for Custom Types

OK, all this fancy talk about object oriented stuff, but I said I wouldn't try to convince you to do everything with OOP. Indeed, how can custom types be used to benefit your average simple Lasso web site?

Most web sites that are using dynamic data in any fashion typically have several pages which allow the display, addition, updating, and deletion of database records. Most novice and even intermediate programmers tend to spread the code for those functions across many files. As the site gets bigger and more complex, code is split out into includes, then the includes start to get a bunch of conditional code to account for "special cases" of various pages.

Custom types can be used to rein in the sprawl and gain some control. For example, if you have news releases on your site. You likely have pages for visitors to search, list, and display details. Likewise there's some admin pages which have similar functions but allow you to edit the records. A news release is a perfect candidate for a custom type. Gather all the routines you have already to search, to select, to add, and update news release records. Take those routines and make each unique one a separate member tag. Take all the variables you use to keep track of the

news releases and make them properties (instance variables) of the custom type.

The advantage of doing this, if nothing else, is the consolidation of all code that manipulates news release records into a single place. Whether you have Hot News sound bites on the home page, or a search for releases needing approval in the admin section, the consolidation of all the code into a custom type will allow you more opportunity to share common code, make it easier to edit all routines if changes to the database are needed, and by using the `$object->task` syntax in your code, it will be easier to read too.

Rearranging your code into ctypes like this also allows you to load the ctype files from `LassoStartup` or use Lasso 8's on-demand loading which keeps the ctype code resident in RAM for faster access. By reducing your dependence on includes, there will be less frequent trips to the hard drive (the slowest service of the server) which will make page processing faster, and your server may be able to handle higher visitor loads.

---

## Wrapup

What I have tried to present is an overview of what custom types are, how they are programmed in Lasso, and some reasons for why they are a beneficial way to write your code.

For people with very small sites of a couple dozen simple pages which rarely change, then I admit using custom types (or even custom tags and includes for that matter) are not very critical. However, for developers constantly creating new sites, and captives frequently enhancing and extending their web sites, the use of custom types can put you on the road of creating highly reusable and reliable code which will save you time and (either make or save more) money.

There are many more aspects to custom types that could be discussed, but that starts to get into some more esoteric aspects of object oriented programming that I promised I would avoid. Some of you have likely noticed that I have not discussed composition, patterns, and several other topics. I have intentionally skipped those topics for this article.

I have also not been very detailed with code examples. Again, that was on purpose. I wanted this article to explain the concepts and provide a high level view of how things are done with custom types. I wanted you to focus on the ideas rather than the details of the implementation. You can get those details from the documentation and `LassoTalk` as you advance.

What I hope you'll do is simply start to view your code from the ctype perspective (discrete processes performed on a shared data set), and get started with consolidating your code and creating some simple custom types. As you build more ctypes, you'll start to see where you can pull

out pieces into shared tags and types. As you experiment and study, it will make you a better programmer. It will make your applications better programs. Ultimately that should lead to happier customers and bosses. 🍀