



AGETOR®

IML Tag Library
User Guide

Contents

Contents	2
Preface	3
Preprocessing the IML document	4
Embedding code using the INSIDE processing instruction.	4
Using <inside> tags directly in HTML code	5
Using the inside attribute in HTML tags	5
Defining and referring macros	5
Declaring methods	6
Declaring variables	6
Retrieving input parameters	8
Retrieving simple types:	8
Retrieving structures:	8
Examples - Security	9
Service components	11
Data components	12
Locating data components	12
Generating data components for IDL modules	12
Writing specialized Data Components	13
Variable values and attribute values	14
Using variable values.	14
Conditioning attribute values.	14
Presentation markup	16
Components	17
Declaring a component	17
Using a component	17
Configuring a component without changing it	19
Extending components	20
Good component making	21
IML Tag Library	22
Attribute condition	22
Special characters & > < / " \	22
Special attributes	24
DTD's	24
The <code>ihhtml</code> tag	25
The <code>parameter</code> tag	26
The <code>variable</code> tag	27
The <code>set</code> tag	28
The <code>value-of</code> tag	29
The <code>space</code> tag	30
The <code>if</code> tag	31
The <code>choose</code> tag	32
The <code>for</code> tag	33
The <code>foreach</code> tag	34
The <code>link</code> tag	36
The <code>component</code> tag	38
The <code>folder</code> tag	40
The <code>barcode</code> tag (optional)	42
Where to find more information	43

Preface

Servlets are server-side Java programs producing HTML output. As opposed to static HTML pages the output produced by servlets are dynamically generated when requested by clients.

Writing servlets can be done "the hard way" by writing Java code that produces HTML. However this is tedious and prevents the usage of HTML editors for designing layout etc. IML is a simple extension to HTML 4.0 that allows embedding Java code in standard HTML or XML documents.

Writing servlets using IML is like writing static HTML except for the dynamic parts of the output. You may use your preferred HTML editor to do the design and layout and then add some extra tags for generating dynamic output. The extra tags are designed so that you are still able to edit and view the HTML in your editor.

This document is aimed to web developers, who want to develop servlets the easy way. It explains how to use all the different tags in the IML tag library, and illustrates each of them with examples.

We assume that you are familiar with HTML, and knows a little bit of Java.

Preprocessing the IML document

The iml2java preprocessor accepts an IML file containing non HTML markup and embedded code as input and produces a Java servlet.

Embedding code in IML is done in accordance with standard HTML syntax using alternatively processing instructions or attributes in other tags. The preprocessor compiles the generated Java source into Java bytecode.

The transformation done by the preprocessor involves insertion of extra Java code and replacing HTML code with print statements, generating the HTML code. This means that line numbers in the generated Java code does not match line numbers in the IML file, so if the compilation of the generated source results in errors, you can not rely on line numbers to locate the errors in the IML file.

It is worth noting that the HTML 4.0 is somewhat stricter than earlier HTML versions, because it is well formed XML.

- Special characters, like Danish letters, must be written using the correct HTML construct.
- All opening tags must be accompanied by a matching closing tag.
- Empty tags must include the terminating slash, as in `<tag-name/>`.
- All tags must be nested correctly.
- One and only one top level tag is possible.

Empty
Tag

`<Qty unit="g" />`

The iml2java is based on a standard SAX XML parser and enforces these rules. The IML document should start with the following XML instruction:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes" ?>
```

The version tag indicates the version, and the given encoding allows you to use international characters. The standalone tag may be omitted but indicates that this document does not depend on other documents.

Embedding code using the INSIDE processing instruction.

Java code is embedded in XML processing instruction blocks as shown below. The block must start with the processing instruction opening tag "`<?>`" immediately followed by the keyword "inside". All Java code before the processing instruction closing tag "`?>`" is left unchanged by the preprocessor and eventual syntax or other errors will be fed to the Java compiler.

```
<ihtml>
<body>
<?inside
    out.println("The current date is " + new Date());
?>
</body>
</ihtml>
```

The example above will produce the following servlet code:

```
out.println("<html>\n");
out.println ("<body>\n");
    out.println("The current date is "+ new Date());
out.println ("</body>\n");
out.println ("</html>\n");
```

When this output has been fed through a Java compiler a servlet will be produced. This servlet will generate a HTML document containing the current date.

This syntax for embedding Java code is appropriate for larger code blocks. Single statements are easily embedded using the `<inside>`-tag as a normal HTML tag, outside processing instruction blocks.

Using `<inside>` tags directly in HTML code

`<inside>` tags can be inserted directly in the HTML code, outside processing instruction blocks. Since the tag is not part of HTML it will be ignored by most HTML browsers/viewers. This allows static values, for instance data values in table cells, to be replaced by dynamic content, when the servlet is executed.

```
<td width="180" ><font size="-1">
<inside inside="out.println(customer.name);">John Doe</inside>
</font></td>
```

The above table cell definition contains an `inside` tag which contains the text "John Doe". This text will be shown in the HTML browser when viewing the static IML file, since the `inside` tag is ignored. After preprocessing the text, it will be replaced with the value of the `inside` attribute:

```
"out.println(customer.name);".
```

Using the `inside` attribute in HTML tags

The previous section describes how to use the `<inside>` tag to replace static content with dynamic content.

The value of the `inside` attribute contains the code that produces the dynamic code. However the `inside` attribute applies to any HTML tag and is not restricted to the `<inside>` tag, and may be used to generate attributes to HTML tags.

The example below extends the previous example by showing how to produce the `width` attribute for the `<td>` tag dynamically from an embedded Java variable named "width". Note the use of backslashes before quotation marks within Java strings:

```
<td inside='out.println("width=\""+width+"\"");' <font size="-1">
<inside inside="out.println(customer.name);">John Doe</inside>
</font></td>
```

Further the `inside` attribute may contain a macro reference to a previously declared code block.

Defining and referring macros

An IML processing instruction block may contain a macro definition of a Java code block, which can be referred to by name. This code block does not produce code when defined, but each time it is referred the code is produced.

Referencing a macro is possible in all HTML tags after the definition, by including the macro attribute in the tag with the name of the code block.

```
<?pokecolor
  if(x == true) {
    out.println("bgcolor='#FFDDAA'");
  } else {
    out.println("bgcolor='#00DDAA'");
```

```
    }
?>
<html>
<body>
<table><tr macro="pokecolor"><td>
</td></tr><table>
</body>
</html>
```

The macro attribute tells the preprocessor that the string value contains the name of a previously defined macro. In the example above the macro attribute tells the preprocessor to insert the code block named "pokecolor". Any HTML before or after the macro attribute is left untouched by the preprocessor.

This allows the IML file to be previewed and edited using standard HTML tools. The <inside> tag and macro attribute are ignored and not shown.

Declaring methods


Declaring methods that can be called from within the document is done in a method processing instruction. The following example uses a HTMLList class to generate a list of customers and invokes a method to make the parameters.

```
<html template="dk.bording.idl.vss.t">
All customers:
<?inside
    Vector customers = getCustomers();

    new dk.bording.servlet.HTMLList(customers) {
        public void printElement(Object element) {
            Customer customer = (Customer)element;
            out.print(customer.name);
        }
        public String getElementLink(Object element) {
            Customer customer = (Customer)element;
            return "servlet.editCustomer" + makeParameters(customer).urlencode();
        }
    }.print(out);
?>
<?method
    ServletParameters makeParameters(Customer customer) {
        ServletParameters sp = new ServletParameters(null);
        sp.putString("id",customer.id);
        sp.putString("operation", getCustomerText);
        return sp;
    }
?>
</html>
```

Any number of method processing instruction blocks may be declared and each may contain several method declarations. Methods can only operate on parameters passed to the method. Methods may be declared anywhere in the document, and can be called prior to declaration.

Declaring variables

 NOTE: This feature should be used with extreme caution, and is likely to be deprecated in future releases.

It is also possible to declare variables/fields globally (i.e. within the generated class) either in the methods template section or in a global template section both after the final `</html>` tag.

- ⚠ NOTE: Care should be taken when assigning values to global variables. Do not assign variables defined as parameters in IDL, because these will only be instantiated when the body of the document is processed. So accessing these can produce `NullPointerException` during instantiation of the servlet object, hence preventing the servlet from running at all.

Retrieving input parameters

Parameters to your IML program typically comes from a user submitting a form or clicking an URL on a web page. We will call these the *Servlet input parameters* (since your IML eventually becomes a servlet) or simply *parameters* or *input*. A number of methods to access these parameters exists.

Retrieving simple types:

```
<type> d = getServletParameters().get<type>Parameter("d");
```

This extracts the parameter named *d* to the variable of the same name. If this parameter is an integer the `<type>` should be `Int`. The code below gets an int and a string using this method.

```
int district;
String name;
district = getServletParameters().getInt("district");
name = getServletParameters().getString("name");
```

Retrieving structures:

If a complex type has been defined in IDL it may contain simple and nested complex types. Usually you would want to build an instance of such a type from the input. This may be achieved by using the `extractParameters` method that is automatically provided with the type you defined in IDL. This method takes three parameters: the name of the complex object you are extracting, the servlets parameters (obtained by the `getServletParameters()` method) and the orb that contains the type you want to extract.

```
<type> t;
t.extractParameters("t", getServletParameters(), myOrb);
```

This invocation will extract all the parameters from the input with names matching the names in the type of *t*. The instance *t* will contain the extracted values. The following rules apply to the input parameter naming:

If *a* is a member variable of some complex structure it should be fully qualified when referred.

If *v* is some recursive structure (a type defined in IDL using the `sequence` directive) the individual elements should be specified and referred using brace notation `[]`.

For example, assume the complex type `MyType` is defined as follows:

```
struct MyType {
    long a;
    string b;
    MySequenceType c;
};
```

Values are assigned to a parameter named *t* of type `MyType` by the following syntax:

```
t.a
t.c[4].b
```

The type `MySequenceType` contains a complex type with a member named *b*.

Examples - Security

This section describes how to restrict the access to an IML-based system. The method invocations described earlier are translated into internal requests to the webask broker which in turn routes the request to an adequate daemon. By default the system will try to login at the broker as user AGETOR_ANONYMOUS with password AGETOR_ANONYMOUS_PWD on the first method invocation. If these properties are not set, no methods will be accessible without an explicit login (see below). The AGETOR broker is capable of checking if a user is allowed to make a request and will compare the user's access rights against the permissions of the requested method. To benefit from this facility you need to login at webask with a username, environment and password using the `brokerLogin()` method. On success this method will store the user information along with the *session number* returned from the broker. Generally the `setUserInfo` method may be used to store persistent information across IML-programs. On a later invocation of another IML page this information may be obtained using the method `getUserInfo`.

Below a login program is shown. This program expects the username and password as parameters from a submitted form. If the login is successful the user is redirected to one page, if not to another.

```
<!-- INSIDE
<inside>
#template<ServletLogin.t>
#template.body
</inside>
-->
<html> <head>

<!-- INSIDE
<inside>
/**
** This program expects the parameters 'user' and 'pwd'.
** Using these it tries to obtain a sessionnumber from the broker
** redirecting to badPage on failure. On success the session number
** username environment and password are stored as persistent
** userinformation for the user.
** The redirection is done using httpd-equivalent tags in the
** head part of the returned html-page. This seems to work well.
**/

ORB orb;
int sessno=-1;
boolean failure = false;
String badPage="http://www.altavista.com";
String goodPage="http://www.muzikalite.net";
String urlstr=goodPage;

String server=System.getProperty("AGETOR_BROKER_HOST");
int iPort=new Integer(System.getProperty("AGETOR_IPORT")).intValue();
String env = System.getProperty("AGETOR_ENV");

// get user and password and try login
String user = getServletParameters().getString("user");
String pwd = getServletParameters().getString("pwd");
try {
    orb = createORB(server, iPort);
    sessno=brokerLogin(orb, user, env, pwd);
} catch (RemoteException re) {
    failure=true;
    urlstr=badPage;
```

```
    }

    // redirect user to appropriate page using html meta-equiv tag
    out.println("<META HTTP-EQUIV=\"refresh\" CONTENT=\"0 ;url="+
urlstr+"\" >");

</inside>
-->
</head><body></body></html>
<!-- INSIDE
<inside>
#template.end
</inside> -->
```

Service components

Service components are the preferred way of invoking service methods from a servlet. Service components are Java objects holding the parameters to each method in an interface as variables. This frees you from explicitly declaring the parameters for a method invocation.

```
<inside:service name="dc1" home="dk.bording.idl.test.TestHome"  
class="dk.bording.idl.test.TestData"/>
```

This declares a service component named dc1 implementing the IDL interface Test specified by module dk.bording.idl.test, and allows you to invoke the methods defined in it.

Data components

Maintaining state across several servlet invocations is possible through datacomponents. An IML document registers a datacomponent and then have access to the objects of this component. The datacomponent can live for as long as the WebServers servletrunner.

Locating data components

Using the datacomponent tag the IML can access datacomponents:

```
<inside:datacomponent sessionscope="global" servletscope="global" name="dc1"
home="dk.bording.idl.test.TestHome" class="dk.bording.idl.test.TestData"/>
```

- **name** (required): The name of the datacomponent instance declared in your servlet.
- **home** (required): The fully qualified class name of the Home interface for the datacomponent.
- **sessionscope** (default=local): Defines whether the datacomponent is shared across all sessions (global), or local to this session.
- **servletscope** (default=local): Defines whether the datacomponent is shared between all IML servlets or local to this IML document.
- **class** (optional): Defines the type of the datacomponent instance in your servlet. The class defaults to a generic datacomponent, so specifying the class allows you to avoid casting the datacomponent to the correct type.

Accessing shared datacomponents across all session and/or all IML servlets requires you to specify the correct instance name of the datacomponent. Omitting the sessionscope and servletscope retrieves a local session and servlet datacomponent. When datacomponents are local to a session, it means that no other users will be able to interact with the datacomponent instance, i.e. it will be private to this session. The servletscope defines whether the datacomponent instance is accessible from other servlets or only from this servlet.

Sessionscope Servletscope	local	global
local	Instance is private.	Instance shared across sessions.
global	Instance shared across servlets.	Instance shared across sessions and servlets.

Generating data components for IDL modules

Quite similar to servlet templates, each IDL module generates a data component containing the parameters of each operation in each interface as variables.

Using datacomponents does not require you to use a servlet template (however this is possible). Simply omitting the template attribute of the html tag will cause the iml2java tool to base the servlet on a generic template.

Writing specialized Data Components

Instead of using Data Components generated from IDL you can create your own specialized Data Components by writing a Java class.

You can do this by extending an `AbstractDataComponent` which does not contain any variables, and you can also subclass an IDL generated Data Component.

To extend `AbstractDataComponent` you must write a Java class containing code similar to:

```
package mypackage.datacomponent;

import java.util.*;
import dk.bording.inside.servlet.datacomponent.AbstractDataComponent;

/**
 * MyDataComponent is a specialized DataComponent containing a Vector of names.
 */
public class MyDataComponent extends AbstractDataComponent {

    private Vector names;

    public MyDataComponent(StructFactory fac) {
        super(fac);
        names = new Vector();
    }

    public void addName(String name) {
        names.addElement();
    }

    public Enumeration getNames() {
        return names.elements();
    }
}
```

This very simple Data Component allows you to add names to it and enumerate all the added names at some later time. The names added to the Data Component will be available across several invocations of the same servlet by the same user, with default scoping rules.

Variable values and attribute values

Using variable values.

This is not possible because this is not valid IML:

```
<inside:variable name="dynamicnumber" type="int" value="20"/>
<table width="<value-of name="dynamicnumber"/>">
```

instead the AGETOR Development Kit provides you with a powerful feature:

```
<table width="{dynamicnumber}">
```

This maybe not look very impressive but actually gives you a lot of possibilities. With this tool/feature you can assign dynamic values to attributes or you can use a variable in more than one place, and thereby by changing the value of the variable it will affect the values of the attributes where this variable is used.

Example:

By changing the value of the dynamicnumber below, the changes will affect not only the table attribute width but also the name of the picture in the image src attribute.

```
<inside:variable name='dynamicnumber' type='int' value='20' />
<table width='{dynamicnumber}'>
  <image src='picture{dynamicnumber}.jpg'>
  </image>
</table>
```

NOTICE: Because that the dynamic assignment of variables is evaluated as a string the result of the following code:

```
<inside:variable name='dynamicnumber' type='int' value='6' />
<table width='{dynamicnumber+1}'>
</table>
```

results to: `<table width='61'>` and NOT `<table width='7'>` which you might expect.

If you want to have the dynamic expression evaluated as a numeric expression, you have to put the expression in parentheses to mark it as one single expression, like this:

```
<inside:variable name='dynamicnumber' type='int' value='6' />
<table width='{ (dynamicnumber+1) }'>
</table>
```

this evaluates to: `<table width='7'>`.

Conditioning attribute values.

Yet another feature is available in the range of features provided with the adk, and this one allows you to put conditions on attributes.

In its basic form it looks like this:

```
<table width='{condition}?{then}:{else}' />
```

The condition, which can be a boolean or a valid java-expression which evaluates to a boolean, is the condition to evaluate. If the condition evaluates to true the {then} part is used as the attribute value otherwise the {else} part is used.

When using numbers or variables in this construction they must be enclosed in double quotations:

```
<table width='{dynamicnumber>10}?{25}:{0}' />  
<table width='{dynamicnumber>10}?{number1}:{number2}' />
```

If you use strings in the construction they must be enclosed in single quotations:

```
<img src='{dynamicnumber>10}?{"path to graphic"}:{"another path"}' />
```

Notice: some attributes are treated special and have their own kind of conditioning, therefore conditioning them using the above construction makes no sense. See [Special attributes](#).

Presentation markup

This section describes the available presentation markup within IML.

Presentation markup allows you to use generic HTML layouts without lots of HTML code in your IML document.

Components

This section describes how to declare and use components.

See also `element component`, `element property`, `element section` and `element invoke`.

Declaring a component

When declaring a component you have to add a `type` attribute to the `ihhtml` tag, so that it looks like this:

```
<ihhtml type="component">.
```

Then you may declare an unlimited number of properties and sections. If the component does not need any properties or sections to add easily changeable functionality, then do not add any. It is completely up to you.

To add a property write:

```
<inside:property name="<name-of-property>" type="<type-of-property>"/>
```

To add a section write:

```
<inside:section name="<name-of-section>">  
  ... markup, text, anything ...  
</inside:section>
```

After you have declared your properties and written your sections you can add the component functionality. Here you can use the declared properties.

Example:

```
<ihhtml type="component">  
  
<inside:property name="border" type="int" default="0"/>  
<inside:property name="age" type="int"/>  
<inside:property name="person" type="String" default="Stranger"/>  
  
<inside:section name="printHello">  
  <b> Hello </b> <inside:value-of name="person"/>  
</inside:section>  
  
<table cellspacing="2" cellpadding="3" border="{border}">  
  <tr>  
    <td><b>Hello World!</b></td>  
    <td><b><inside:value-of name="person"/></b></td>  
  </tr>  
</table>  
  
</ihhtml>
```

Using a component

To use a component use the `inside:component` tag. If the properties in the component have not been given a default value, remember to assign a value to them. The component type is the name of the component IML file.

Example:

```
<ihtml>

<head>
<title>Test</title>
<meta name="GENERATOR" content="Inside"/>
<style></style>
</head>
<body>

<inside:component name="reusableComponent">
  <inside:property name="border" value="10"/>
  <inside:property name="age" value="25"/>
  <inside:property name="person" value=' "young and handsome" ' />
  <inside:invoke name="printHello"/>
  <inside:invoke name="printHello"/>
  <inside:invoke name="printHello"/>
</inside:component>

</body>
</ihtml>
```

Configuring a component without changing it

Sometimes you want to change the behaviour of a component or add something to it without changing the component itself. This can be done in two ways, one way is overwriting a section and another way is using a component to configure a component.

Overwriting a section

If you do not want to change an entire component it is also possible to overwrite a section. To overwrite a section you need to know the name of the section you want to overwrite:

```
<inside:component name="reusableComponent">
  <inside:section name="printHello">
    Goodbye!
  </inside:section>
  <inside:property name="age" value="25"/>
  <inside:property name="person" value="young and rich"/>
  <inside:invoke name="printHello"/>
  <inside:invoke name="printHello"/>
</inside:component>
```

Now when invoking the printHello-section it prints out "Goodbye!" instead of "Hello!".

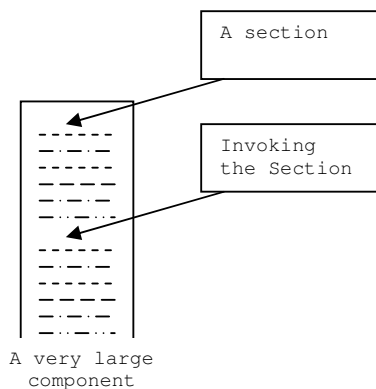
Notice that overwriting a section does not have any effect outside of the use of the component. For instance if you want to use the same component twice in your IML document and you overwrite a section the first time you use the component then the section is not overwritten the second time you use it.

If you have a very large component with lots and lots of IML tags which is very hard to navigate through then you can, instead of changing the component, use a section in the document and then overwrite the section when using the component.

Using sections when making very large components

```
<ihtml type="component">
<inside:property name="str" type="String" default="Empty"/>
<inside:section name="easyToFind">
  This can otherwise be hard to find.
</inside:section>
```

```
... huge amounts of ihtml tags ...
  <inside:invoke name="easyToFind">
... even more ihtml tags ...
</ihtml>
```



Overwriting sections when using a component

```
<inside:component name="veryLargeComponent">
  <inside:section name="easyToFind">
    This is very easy.
  </inside:section>
</inside:component>
```

When you have overwritten the section then it is the new section that will be invoked from within the component. In this way you can easily change the output within the component without changing the component itself.

Using a component to configure a component

Instead of overwriting a section you can use a configurator component inside a configurable component and then just change the configurator component to make the changes take effect in the configurable component. This way is used when you need to have access to variables outside the component in which you want changes to take effect.

The configurable component:

```
<!-- This component uses the configuratorComponent -->
<ihtml type="component">
<inside:variable type="String" name="aVariable" value="Default Name"/>

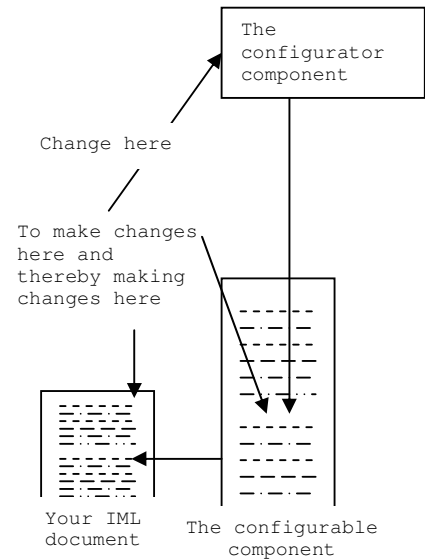
... huge amounts of IML tags ...
<inside:component name="configuratorComponent">
  <inside:property name="name" value="aVariable"/>
</inside:component>
... even more IML tags ...

</ihtml>
```

The configurator component:

```
<ihtml type="component">
<inside:property name="name" type="String"/>

  <h1> <b> <inside:value-of name="name"/> </b> </h1>
</ihtml>
```



Using the configurable component:

```
<ihtml>
... A lot of tags ...
<inside:component name="configurableComponent"/>
... More tags ...
</ihtml>
```

By using this approach you can split components up in more than one component and thereby make the handling and changing of components easier and instead of adding a tag between line 560 and line 561 in a large component you can now add it at line 5 in a small component and gain the same effect.

Extending components

With the ADK 1.2 it is possible to extend components and thereby reuse the functionality of other components without rewriting or copying any code. When you extend components you can invoke the sections and use the properties from the component you are extending. It is not possible to reuse the contents of the component. If you want to reuse some content, put it in a section and use the section.

Examples:

```
<!-- Making a component -->
<!-- This is the extendableComponent -->
<ihtml type='component' >
```

```
<inside:property name='nameString' type='String' default='no name' />
<inside:property name='age' type='int' />
<inside:section name='reuse'>
  <inside:value-of name='nameString' />
  ... Here you can put markup, which you want to reuse ...
  ... It is also possible to put components here ...
</inside:section>
Any markup you put outside a section, can not be extended.
It is simply deleted. So if you need to extend markup,
put it in a section and invoke the section.
</html>
```

```
<!-- Extending a component -->
<!-- This component extends the extendableComponent -->
<!-- This is the extendedComponent -->
<ihtml type='component' extends='component.extendableComponent'>
  <inside:value-of name='age' />
  <inside:invoke name='reuse' />
</ihtml>
```

```
<!-- Using the extended component -->
<!-- This example uses the extendedComponent -->
<ihtml >
  <inside:component name='extendedComponent'>
    <inside:property name='age' value='20' />
    <inside:property name='nameString' value='my name' />
  </inside:component>
</ihtml>
```

By extending a component you can easily reuse the sections and properties of a component. Notice the difference between extending a component and using a component within a component. When you use a component within a component, you are using the output of the component but when you extend a component you use what a component is able to do, not the result of it.

Good component making

When making component dependencies it is very important to comment these dependencies thoroughly so that it clearly states what happens and where it happens if you change the component:

```
<!-- Changes to this component will affect the notOftenUsedComponent -->
<!-- This component uses the veryOftenUsedComponent -->
<!-- By overwriting the printHeadline section you can determine the headline -->
```

By doing this it is easy to see whether a component is dependent on other components and what the changes will affect and also which section overwriting possibilities there are.

Be careful when extending components and using components within components, that you do not lose yourself in detail, but remain in a general view of what dependencies exist between the components you use.

IML Tag Library

IML documents are processed into servlets, and within this process inside transformations may be applied. This enables you to use non-HTML markup both for control and presentation.

To enable Inside transformations, your document must have the root node `<ihtml>`.

Control markup allows you to iterate through dynamic data without any embedded code.

All markup tags are within the inside namespace, so you must prefix `inside:` to all tags. I.e. using the if tag requires you to write `<inside:if ...>`.

In the following we will describe every tag in the IML tag library. Every description contains a table explaining the use of the tag's attributes (if any) and one or more examples. If a tag contain nested tags inside it, it also includes a description, an attribute table and examples for each of them.

The attribute table contains 4 columns:

Attribute: the name of the tag's attribute.

Requ: indicates if this tag is required (✓) or optional.

Java: indicates if this attribute's value may be a Java statements (✓), or if it must be a fixed string.

Description: explains what this attribute does.

But first we describe the use of the condition attribute, special characters, special attributes and DTDs.

Attribute condition

The attribute `condition` allows you to write conditional markup.

Attribute condition	
condition	The condition deciding whether the element is output.

On any element adding the condition attribute will output the element only if the expression evaluates to true. Any nested markup and text will always be output.

Examples:

```
<conditional-element condition=' java-expression'>
  ... markup ...
</conditional-element>
```

```
<a href='{person.website}' condition='!person.website.equals("")'>
  Name: <inside:value-of name='person.name' />
</a>
```

Special characters & > < / " `

Because `iml2java` and most other parsers recognize some special characters as a part of the document definition, you are not able to use these characters directly in your document. E.g. the following statement is not possible: `<inside:if test='rows<10'>` because the less than sign marks the start of the `inside:if` element, this will be interpreted as the start of a new element within the `inside:if` element, and this is logically not possible.

If you need to out print one of the special characters, you are not able to write the character itself, you have to write the character reference.

Description	Character	Reference	As decimal	Special notice
And	&	&	&	
Less than	<	<	<	
Greater than	>	>	>	
Apostrof	'	'	'	
Quotation mark	"	\"		Can not be used within an Inside element.

Example:

```
<inside:variable name='myString' type='String' value='me&amp;you' />
<inside:if test='someInteger&lt;10'>
  You can write iml2java&apos;s \"special\" characters.
  <inside:value-of name='myString' />
</inside:if>
```

Special case: When you are writing JavaScript within your document, it is often necessary to use quotation marks. Within the <script> tag it is possible to write the quotation mark directly, but outside the <script> tag you have to use the quotation mark reference.

Example:

```
<input type='text' name='{varName}' size='10' onMouseOver='return
calculateScript (\">{anotherNumber}\")' />
```

Example:

```
<IHTML>
<HEAD>
  <SCRIPT>
    var myPage;
    function openMyPage() {
      if (!myPage || myPage.closed){
        myPage = window.open("myPage.html", "_blank");
      } else {
        alert("MyPage.html is already opened");
      }
    }
  </SCRIPT>
</HEAD>
<BODY>
  <FORM>
    <INPUT TYPE="button" onClick="openMyPage();" VALUE="open MyPage">
  </FORM>
  <A HREF="http://www.bording.dk/ "
    onMouseOver='this.style.fontWeight=&apos;bold&apos;;'
    onMouseOut='this.style.fontWeight=&apos;normal&apos;;'>
    Go AGETOR!
  </A>
</BODY>
</IHTML>
```

Notice: The special characters have to be used with cautious. You have to be absolutely sure that one of these characters is needed before they are used because they can conflict with the parsing of the document. When in doubt do not use any special characters.

A complete list of available references can be found at:
http://www.utoronto.ca/webdocs/HTMLdocs/NewHTML/iso_table.html

Notice that in most cases the decimal number of the character can be used instead.

Example:

```
<inside:if test=' decimals'>
  You can write references as hexadecimals on the form
  &#123; ampersand followed by a # and the number of the character
  and ending it with a semicolon &#125; .
</inside:if>
```

Special attributes

The following attributes are treated special:
checked, nowrap, selected, multiple, noresize

Because of differences between html versions the following start element is not the same in all browsers: `<table nowrap='false'>`
Older browsers will evaluate 'nowrap' to be true, because it is present. Only newer browsers evaluate 'nowrap' to be false because they look at the value of 'nowrap'. Iml2java suppresses the special attributes if its value is false and only if the value evaluates to true, it is included in the as an attribute in the element.

The value of the above attributes needs to be a java-expression, which evaluates to a boolean.

Example:

```
<table nowrap='true'>
<table nowrap='str.equals("AGETOR")'>
```

The expression is evaluated at runtime.

DTD's

When writing text again and again you can define document type definitions in top of your IML document and use the entities in the dtd again and again.

Example:

```
<!DOCTYPE dtd [
  <!ENTITY in 'AGETOR from Bording Data A/S'>
]>
<ihtml>
<inside:for stop='20'>
  &in;
</inside:for>
</ihtml>
```

The `ihtml` tag

This tag indicates that this document is an IML file, and is called the root tag. The root tag is used to determine what kind of markup language is used, so by changing the root tag you can adapt the IML parser to all markup types (e.g. `html`, `xml`, `wml`, etc). The contents of an IML-document default to HTML, so if you write e.g. `xml`, remember to add the `contenttype` attribute to your root tag.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
<code>imports</code>			The Java packages that you want to be able to access in your IML document.
<code>contenttype</code>			Defaults to "text/html".
<code>type</code>			Defaults to "servlet" (see The <code>component</code> tag).

Examples:

```
<ihtml imports='dk.bording.inside.util.*;java.util.SomeClass'>
  ... content ...
</ihtml>

<xml type='component' contenttype='text/xml'>
  ... content ...
</xml>
```

The parameter tag

This tag defines a variable, then extracts the variable's value from the URL parameters of the form `http://somepage?<parameter_name>=<parameter_value>`. If the URL does not contain the parameter, a default value can be used.

This tag must be placed as one of the first tags in the file.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓		The variable's name. This name must match the name of the URL parameter it gets its value from.
type	✓		The variable's type (the only types supported at this time are: int, long, double, boolean, short, float, String and Date).
default		✓	Determines a default value for the parameter, if it is not given by the URL.

Example:

The following tag will extract the string `language` from the URL. If it's not found, the parameter value will be "danish":

```
<inside:parameter name='language' type='String' default='danish' />
```

The variable tag

The `variable` tag allows you to declare variables and initialize them.

The `variable` tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓		The name of the variable to declare.
type	✓		The type of the variable. If the type needs a constructor to be called (exceptions are <code>String</code> and primitive types), the value must be <code>'new'</code> .
value		✓	The initial value assigned to the variable. It must be <code>'new'</code> for variables that need to call a constructor. Then the constructor's parameters must be given as the contents of the markup.

Examples:

The following markup shows how to use the `variable` tag for declaring a `String` variable:

```
<inside:variable name='str' type='String' value='some text' />
```

The following markup shows how to use the `variable` tag for declaring a variable of a primitive type:

```
<inside:variable name='number' type='int' value='42' />
```

The following markup shows how to use the element `variable` for declaring a `String` variable, and assigning the value from the contents:

```
<inside:variable name='str' type='String'>This is the value.</inside:variable>
```

The following markup shows how to allocate new variables of a type that needs a constructor:

```
<inside:variable name='date' type='Date' value='new' />
```

The following markup shows how to allocate new variables of a type that needs a constructor with a parameter:

```
<inside:variable name='object' type='Integer' value='new'>57</inside:variable>
```

⚠ Currently this only works when no new lines separate the start tag from the end tag.

The set tag

The element `set` allows you to modify the value of declared variables.

The set tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓		The name of the variable to assign the value.
value	✓	✓	The initial value assigned to the variable.
condition		✓	If this condition evaluates to false, then this tag is simply ignored.

Examples:

The following markup shows how to use the `set` tag for modifying an int variable:

```
<inside:set name='number' value='number+1' />
```

The following markup shows how to use the `condition` attribute. The number is only set when `i` equals 0:

```
<inside:set name='number' value='number+1' condition='i==0' />
```

The value-of tag

This tag prints the value of a variable in the HTML page. The `value-of` tag can also format decimal numbers and date values. The date format syntax is the same as the one used in `java.text.SimpleDateFormat` (see its [API Specification](#) for a detailed description).

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓	✓	The name of the variable. It may be a simple variable, an IDL structure or a Java method invocation returning a value. The variable must be previously declared (see the <code>variable</code> tag).
decimal		✓	Number of digits to show on the decimal part of a number. The number is rounded. The variable must be of type <code>double</code> .
todateformat		✓	The format to use when printing the date.
fromdateformat		✓	The format the date variable has. The default is the <code>java.util.Date</code> format (EEE MMM dd HH:mm:ss zzz yyyy).
condition		✓	If this condition evaluates to false, then this tag is simply ignored.

Examples:

Assuming `var` is the number 4.256, the following tag will print it out as 4.26:

```
<inside: value-of name='var' decimal='2' />
```

Assuming `dat1` is the date `new java.util.Date(101, 1, 26)`, the following tag will print it out as 26.02.2001:

```
<inside: value-of name='dat1' todateformat='dd.MM.yyyy' />
```

Assuming `dat2` is the String 270200, the following tag will print it out as 27.02.2000:

```
<inside: value-of name='dat2' todateformat='dd.MM.yyyy' fromdateformat='ddMMyy' />
```

The space tag

The `space` tag allows you to force the writing of a space character. Iml2java removes spaces by default. If you want all spaces preserved in the whole document, you can set the Inside property: `inside.tools.impl2java.whitespaces`.

The `space` tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
no		✓	The number of spaces, which must be forced at this location. The default is 1.

Example:

The first example will have only one space between `'Name:'` and `'Jens Jørgensen'` in the generated document, as the second example forces 10 white spaces:

```
Name:      Jens Jørgensen
```

```
Name:<inside:space no='10' />Jens Jørgensen
```

The `if` tag

This tag prints its body only if its `test` attribute evaluates to true.

This tag contains the following attribute:

Attribute:	Requ:	Java:	Description:
test	✓	✓	A boolean expression.

Example:

```
<inside:if test='boolean-expression'>
    This will only be evaluated if the boolean-expression
    evaluates to true.
</inside:if>
```

The choose tag

This tag allows you to choose one of several bodies, providing a condition for each of them. The first condition that evaluates to `true` will have its body printed out, and all the others will be ignored. If none of the conditions evaluates to `true`, the default body (`else` tag) will be printed out. There can only be one `else` tag within a `choose`, but an unlimited number of `when` tags are allowed. There is no point in making a `choose` tag with no `when` tags within.

The `when` tag contains the following attribute:

Attribute:	Requ:	Java:	Description:
test	✓	✓	A boolean expression.

Example:

```
<inside:choose>
  <inside:when test='boolean-expression'>
    .. chooses this if the test evaluates to true ..
  </inside:when>
  <inside:else>
    .. chooses this if no tests evaluates to true
  </inside:else>
</inside:choose>
```

The `for` tag

This tag allows iteration of its body. It starts at the `start` value and stops just before the `stop` value. The index used to count the iteration can also be used in the tag's body (`indexname`). If the stop and start variables are numbers it is possible to run from start to stop decreasing the start value. Other wise it is always increasing the start value from start to stop.

Notice! The default indexvalue “n” exists no more and is there for only available if it is specified as your `indexname`.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
start		✓	The first value of the iteration. If it is omitted, the default value 0 (zero) is used.
stop	✓	✓	The value where the iteration stops. This value will not be included in the iteration.
indexname			Variable containing the iteration's index value.

Example:

The following tag will print its body 3 times after each other (result: 1, 2, 3):

```
<inside:for start='1' stop='4' indexname='y'>
  <inside:value-of name='y' />,
</inside:for>
```

The following tag will print its body 5 times after each other (result: 5,4,3,2,1):

```
<inside:for start='5' stop='0' indexname='y'>
  <inside:value-of name='y' />,
</inside:for>
```

The foreach tag

This tag allows iteration of sequences. Supported types of sequence are `java.util.Vector`, `java.util.Hashtable`, `java.util.Iterator`, `com.sun.java.util.collection.Iterator` and `java.util.Enumeration`. It runs through the sequence of elements, performing its body one time for each element in the sequence. All elements in the sequence must be of the same type.

The sequence's iteration can be sorted by one or more attributes of the sequence's elements. For example, if the sequence has elements of the following type:

```
public class Element {
    private String name, secret;
    public double amount;

    public String getName() {
        return name;
    }
}
```

then the iteration can be sorted by name or amount, but not by secret, as it is a `private` variable and it does not have a `get` method.

The `sequencetype` variable should be defined like this:

`java.util.Vector` as `Vector`

`java.util.Hashtable` as `Hashtable`

`java.util.Enumeration` as `Enumeration`

`java.util.iterator` as `java.util.Iterator`

`com.sun.java.util.collection.Iterator` as `Iterator` or `com.sun.java.util.collection.Iterator`

It defaults to `Vector` but we recommend that you always define the variable.

Notice: The for-each tag do not support the default `indexname` "n" no more. To use "n" as a `indexname` you must specify it, in your tag.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
Name	✓		The name of the variable holding an element from the sequence at each iteration. This variable will be declared for you.
Class	✓		The full class name of the sequence's elements.
Sequence	✓	✓	The name of the variable containing the sequence.
Sequencetype			The type of the variable containing the sequence. Default is <code>Vector</code> .
Indexname			Variable containing the iteration's index value.
Sort		✓	The names of the attributes to sort the iteration by. The attributes must be separated by comma. Default sorting is ascending. To sort in descending order, add <code>desc</code> after the attribute's name. The given attributes must be <code>public</code> in their class or have a <code>get</code> method. The sort attribute also support the use of inside variables by using this syntax: <code>{variable-name}</code>
Start			If you do not want to go through the whole sequence, you can set this attribute to start at a specific index.
Stop			If you do not want to go through the whole sequence, you can set this attribute to stop before the specific index. This index will not be included in the iteration.

Example:

Assuming that the Vector `v` contains the following elements:

```
Vector v = new Vector();
v.add(new Element("a", 1.0, "s1"));
v.add(new Element("c", 6.0, "s2"));
v.add(new Element("a", 7.0, "s3"));
v.add(new Element("c", 4.0, "s2"));
```

the following tag will print the names in descending order, then ordered by their amounts:

```
<inside:for-each name='elem' class='Element' sequence='v'
                sequencetype='Vector' sort='name desc, amount'>
  <inside:value-of name="elem.getName()" />:
  <inside:value-of name="elem.amount" /><br/>
</inside:foreach>
```

and the result will be:

```
c, 4.0
c, 6.0
a, 1.0
a, 7.0
```

Assuming that the Vector `v` now contains the following Strings:

```
Vector v = new Vector();
v.add("Jens");
v.add("Martin");
v.add("Peter");
v.add("Karl");
```

the tag:

```
<inside:for-each name='elem' class='java.lang.String' sequence='v'
                sequencetype='Vector' indexname='i'
                start='1' stop='3'>
  <inside:value-of name='i' />:<inside:value-of name='elem' /><br/>
</inside:foreach>
```

will result in:

```
1, Martin
2, Peter
```

The link tag

The `link` tag gives you a markup that can verify the parameters, which you want to link to, and easily put the parameters within the URL. This tag corresponds to the HTML `<a>` tag, except that it correctly URL-encodes the link and its parameters for you.

Instead of hard coding links between servlets, the `link` tag allows you to dynamically assign parameters to an URL. The link's name may be a direct link or a logical name, defined in the file `/conf/servlets.cfg` (the link must be defined in this file before compiling the IML file using `iml2java`).

The `servlets.cfg` file will globally affect the name attribute of the link tag at compile time. E.g. if you need to change a link, you only have to change it in the `servlets.cfg` file. But beware that the changes made in the `servlets.cfg` file will only affect those files, which are compiled after you have made the changes. That means that you can have a file with a link to the old link, by being compiled before you made the changes in the `servlets.cfg` file. So be sure to recompile the files where you want the changes to take affect, after you make changes in the `servlets.cfg` file. If you are trying to link to an older IML file which have been written under a former ADK than IDK1.1.7, that IML file will not reflect recent developments, and must therefore be recompiled. After compilation it is now possible to link to that IML file.

Observe that the `parameter` tag inside a `link` tag differs from the previously mentioned `parameter` tag, as it *provides a value* for a URL parameter, instead of *extracting its value* from the URL.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓		The logical name of the servlet to link to.
target		✓	The target frame where to display the link.
type		✓	Determines if it is an Inside link ('inside', meaning that the actual servlet to link to is defined in <code>servlets.cfg</code>) or any other kind of link ('other'). The default is 'inside'.
autoload		✓	If this parameter is set to true, this link will load itself without the user clicking on it. The default is false.
time		✓	The amount of time (in seconds) to wait before autoloading the link. If autoload is set to false, this parameter is meaningless. The default is 0 (zero).

The parameter tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
name	✓		The parameter's name.
value		✓	The parameter's value.
type			The type of the parameter.

Example:

The following tag will print a link to the servlet `myServlet` with the `user_age` parameter containing the value 25:

```
<inside:link name='myServlet' target='_blank'>
```

```
<inside:parameter name='user_age' type='int' value='25' />  
  Go to my servlet!  
</inside:link>
```

The file `servlets.cfg` must include `myServlet`:

```
<SERVLET NAME="myServlet" CLASS="test.myServlet" />
```

The component tag

This tag allows you to use AGETOR components. For a description of use see Components.

The `property` tag allows you to set values for the component's properties. The `property` is also used when defining the component properties.

The `section` tag allows you to declare a section on a component that can be invoked repeatedly. The `invoke` tag, as its name says, invokes the given section. This tag is mostly used when defining a new component.

This tag contains the following attribute:

Attribute:	Requ:	Java:	Description:
name	✓		The component's logical name. This name must be defined in the <code>conf/components.cfg</code> file.

The `property` tag contains the following attributes

(when defining component properties):

Attribute:	Requ:	Java:	Description:
name	✓		The name of the property.
type	✓	✓	The property's type.
default		✓	The default value. If the default value is not given, it must be set when using the component.

The `property` tag contains the following attributes

(when assigning values to properties):

Attribute:	Requ:	Java:	Description:
name	✓		The name of the property.
value	✓	✓	The property's value. Notice when referring a String property, single quotations must be used surrounding double quotations.
condition		✓	If this condition evaluates to false, then this tag is simply ignored.

The `section` tag contains the following attribute:

Attribute:	Requ:	Java:	Description:
name	✓		The name of the section to be defined or overwritten.

The `invoke` tag contains the following attribute:

Attribute:	Requ:	Java:	Description:
name	✓		The name of the section to be invoked.

Examples:

The following tag will print the Extraware toolbar component:

```
<inside:component name='extraware.toolbar'>
</inside:component>
```

The result will be:



The following tag will print the same toolbar component, but with its `undo` property set to `true` (it is `false` by default), and its `space` section replaced by a section containing only an empty table cell:

```
<inside:component name='extraware.toolbar'>
  <inside:property name='undo' value='true' />
  <inside:section name='space'>
    <td width="17" height="29">
      </td>
  </inside:section>
</inside:component>
```

The result will be:



This example shows how to define the component (this must be done in a separate file):

```
<ihtml name="yourComponent">
  <inside:property name="border" type="int" default='0' />
  <inside:property name="stringtext" type="String" default=' "Hello!" ' />
  <table cellspacing="2" cellpadding="3" border="{border}">
    <tr>
      <td><b>Hello World!</b></td>
      <td><b><inside:value-of name="stringtext" /></b></td>
    </tr>
  </table>
</ihtml>
```

Then you can use the previous component in another IML file:

```
<inside:component name='yourComponent'>
  <inside:property name='bordervalue' value='10' />
  <inside:property name='stringtext' value=' "How are you?" ' />
</inside:component>
```

The following example shows how to define and invoke a section in a component:

```
<ihtml type="component">
  <inside:section name="sec1">
    ... markup, text, anything ...
  </inside:section>
  <inside:invoke name="sec1" />
</inside:component>
```

The `folder` tag

This tag creates a HTML folder, allowing your web page to have layers. Every layer is called a tab.

The folder must be inside a HTML form to work. Whenever you select a tab in the folder, the page itself is requested again, and the selected tag's content is shown.

The `tabs` tag allows you to create a sequence of tabs in a folder. This tag is used when you need to create a predefined sequence of tabs, which must look exactly the same. The `tabs` tag is a combination of the `tab` and the `for-each` tags.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
width		✓	The width of the HTML folder.

The `tab` tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
title	✓	✓	The text that is shown on the tab.
icon		✓	The icon that is shown on the tab.

The `tabs` tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
title	✓	✓	The text that is shown on the tab.
icon		✓	The icon that is shown on the tab.
seqname	✓	✓	The name which will be added to the name of the tab and the title of the tab to give each tab an unique name. A field name in the sequence is a common way to get a usable name.
sequence	✓	✓	The name of the sequence, which must be a Java Vector.
class	✓		The class of the elements of the sequence.
vname		✓	The name for accessing the current element in the sequence.
indexname			The name of the variable used to cycle through the sequence. Primarily used to nest folder tabs. The default value of indexname is 'm'.

Examples:

```
<form name='form1'>
  <inside:folder>
    <inside:tab title='private1' icon='image1.gif'>
      Some text.
    </inside:tab>
    <inside:tab title='private2' icon='image2.gif'>
      Another text.
    </inside:tab>
  </inside:folder>
</form>
```

```
<inside:folder>
  <inside:tabs title='Words' sequence='Stringlist' class='String' vname='str'
    seqname='str'>
    ... markup ...
  </inside:tabs>
</inside:folder>
```

The `tab` tag must be at the first level within a folder tag. Bad example (does not work):

```
<inside:folder>
  <inside:if test='test-condition'>
    <inside:tab title='private' icon='image.gif'>
      ... markup ...
    </inside:tab>
  </inside:if>
</inside:folder>
```

The tabs tag must be at the first level within a folder tag. Bad example (does not work):

```
<inside:folder>
  <inside:if test='test-condition'>
    <inside:tabs title='ttitle' icon='timage.gif' sequence='Userlist'
      class='User' vname='u' seqname='u.username'>
      ... markup ...
    </inside:tabs>
  </inside:if>
</inside:folder>
```

(The 'if' tag is at the first level of the folder tag. The tabs tag is at the second level of the folder tag it must be at the first level, therefore this construction does not work)

The barcode tag (optional)

The barcode tag produces printable barcodes on your web page. It supports the following types of barcodes:

- Ean8
- Ean13
- Ean128
- Code39

There are two implementations you can use in the library. Dynamically generated barcode images, or image puzzles consisting of individual images that will make up a barcode. See the Barcode documentation for instructions on creating images dynamically.

This tag contains the following attributes:

Attribute:	Requ:	Java:	Description:
code	✓	✓	The code for the image
type	✓		The barcode type
height	✓	✓	The height of the barcode in pixels (dynamically generated images only)

Example: the tag:

```
<inside:variable name='partNo1' type='String' value='4711' />
<inside:variable name='height' type='int' value='30' />
<inside:barcode code='partNo1' type='code39' height='height' /><br />
```

will result in this barcode image (using dynamic image generation):



Where to find more information

If you have questions about the IML tag library, or other AGETOR products, you can look for answers in the AGETOR Downloadcenter (<http://www.agetor.com>), or write to AGETORsupport@bording.dk.