

# CTL Objects

## Table of contents

1 Overview.....	2
1.1 Concepts.....	2
1.2 Mechanisms.....	4
2 Examples.....	4
2.1 Inheritance.....	4
2.2 Overriding.....	5
2.3 Dispatching.....	6
2.4 Instance data.....	7
2.5 Calling a supertype command.....	8
2.6 Template method pattern.....	9
3 Running commands.....	9

This document introduces how to apply CTL's object oriented programming (OOP) features. Why bother with OOP you might ask? The normal reasons come to mind (inheritance, abstraction, encapsulation, polymorphism) of course but there are some practical reasons, too.

While you can get quite far with a solution that is based on a set of domain specific modules with well parameterized commands, you will eventually face new challenges as you try to scale up your solution. These challenges emanate from several areas:

- **Variation.** Your commands need to run in different environments. You've created parameters but keeping track of the correct values and/or reasonable defaults for each variation gets hard to cope with. You can represent each variation as a CTL object and execute your commands in the context of that object.
- **Detail.** Data-driven code makes for maximum flexibility but what do you do as more and more data is needed by your commands? You can keep adding parameters but this can get unwieldy. Use type "attributes" to create a data namespace and standard model to drive your commands.
- **Coordination.** Breaking your automation into modules and commands helps you divide and conquer the problem at hand. But how do you orchestrate actions across many controllers? Objects give you another dimension to represent control. You can target actions not only to a command in a module but to an object (be it local or remote). Objects give you a much finer grained level of control.
- **Scale.** CTL modules let you decompose your solution logically but what if you want some standard behavior that gets overridden? CTL modules support inheritance enabling you to create base types that offer standard behavior and submodules overriding where needed.

CTL supports several object oriented features that together help meet these challenges and allows you to scale up your CTL solutions.

## 1. Overview

While object-oriented principles are pretty mainstream, there are differences when it comes to implementations and terminology. CTL offers several object-oriented features but it is not an object oriented language, nor is it a complete object oriented runtime. That said, it does offer some fundamental OOP features.

### 1.1. Concepts

The following table relates conventional object-orientation terms to the analogous CTL concept:

Object orientation concept	CTL concept	Notes
Class	Type	A CTL <i>type</i> defines commands (the things it can do) and characteristics (attributes and properties). A <i>module</i> is the software artifact produced from the type definition. CTL establishes the <a href="#">Managed-Entity</a> base type.
Object	Object	An <i>object</i> is a particular instance of a type. Objects are initialized with the type's attributes and properties but can override them with their own. CTL requires that each object have a unique name. The word "entity" is also used to refer to objects that are instances of <a href="#">Managed-Entity</a> .
Method	Command	A type's behavior is comprised by its commands. The software artifact of a command is called its <i>handler</i> .
Message passing	Dispatch	CTL objects invoke each other via an internal mechanism called the "dispatcher" that forwards actions from the sender to the receiver.
Inheritance	Supertype / subtype	Types are arranged in a class hierarchy. A <i>subtype</i> is a specialization of its supertype. CTL allows you to declare your type as a subtype of another type. Subtypes inherit all the commands of its super type. CTL currently supports a single-inheritance model.
Abstract class	Type with role="abstract"	You can declare your types as "abstract" or "concrete". An abstract type is one that cannot have object instantiation while concrete types can. The "concrete" role is default.

Static (class) method	Static command	A command can be set to be "static" and signifies it can work outside of an object context. Normally, this is done for utility commands.
Class fields	Attributes	Type characteristics can be described in terms of "attributes". This is done using the <a href="#">attribute</a> tags in the type.xml

**Table 1: Terminology**

## 1.2. Mechanisms

As described in the documentation's [concepts](#) page, CTL is a framework of modules. A module is a self contained component in the CTL system, responsible for its own data, its own activities, and the integrity of its actions.

Module data is in the form of properties, key/value pairs, while its operations are defined in terms of commands. Modules can inherit commands from their "supertype", override their commands or add new commands.

Users can create instances of types, called objects and call commands in the scope of their instance data. It is the framework's dispatcher that does the lookup for the named command specified by the user, resolve it to a command implementation and object, prepare a data context, and execute the implementation code.

## 2. Examples

The following few sections show how to utilize CTL's object oriented features.

### 2.1. Inheritance

Defining a supertype/subtype inheritance hierarchy is easy. Simply refer to the module name that you want to make your supertype. Commands defined in the supertype are inherited by the subtype and do not need to be re-defined in the subtype's type.xml.

The example below shows a type named "poly" that declares a command "salute". To its right is another type, "greeter", that references "poly" as its supertype. The greeter type declares a command of its own called, "greet". The "poly" type is a subtype of Managed-Entity, the CTL basetype that establishes the ability to execute commands.

Supertype

Subtype

<pre> &lt;type name="poly" role="concrete"       uniqueInstances="true"&gt;    &lt;description&gt;     Says hello.     Offers a salute command.   &lt;/description&gt;    &lt;supertype&gt;     &lt;typereference name="Managed-Entity" /&gt;   &lt;/supertype&gt;   &lt;commands&gt;     &lt;command name="salute" description="offers a salutation" command-type="BsfCommand"&gt;       &lt;script language="jython"&gt;print "hi there"&lt;/script&gt;     &lt;/command&gt;   &lt;/commands&gt; &lt;/type&gt; </pre>	<pre> &lt;type name="greeter" role="concrete"       uniqueInstances="true"&gt;    &lt;description&gt;     Inherits poly's salute command and adds its own called greet.   &lt;/description&gt;    &lt;supertype&gt;     &lt;typereference name="poly" /&gt;   &lt;/supertype&gt;   &lt;commands&gt;     &lt;command name="greet " description="acknowledge a salute " command-type="BsfCommand"&gt;       &lt;script language="ruby"&gt;         print 'nice to meet you', "\n"       &lt;/script&gt;     &lt;/command&gt;   &lt;/commands&gt; &lt;/type&gt; </pre>
---	---

As you probably assume, supertypes should be defined before their subtypes. If you build and deploy a subtype before its supertype, you will encounter errors.

From looking at the examples, you might notice how you can mix command-types, with commands defined in the supertype using one command-type (eg, BsfCommand in jython) and commands in the subtype defined using their own implementation (eg, BsfCommand in ruby). There are several choices for [command-type](#) (Shell, Ant, BSF, Workflow).

## 2.2. Overriding

A subtype can override commands inherited from its supertype. To override a command, simply declare the same named command in the subtype's type.xml.

The example below shows a new type, "barker" overriding the "greet" command.

```

<type name="barker" role="concrete"
      uniqueInstances="true">

  <description>
    Inherits greeter's salute and greet commands
but overrides greet.
  </description>

  <supertype>
    <typereference name="greeter" />

```

```

</supertype>
<commands>
  <!--
    ** override the greet command
    -->
  <command name="greet" description="acknowledge a salute "
    command-type="BsfCommand">
    <script language="groovy"><![CDATA[
      println "CHEERS, GROOVY BABY";
    ]]></script>
  </command>
</commands>
</type>

```

CTL does not require that you preserve the same option set but it is considered best practice not to change semantics nor subtract any options.

It's worth understanding what happens internally at the CTL dispatcher level when it comes to overriding. Roughly these are the steps taken by the dispatcher:

1. Dispatcher receives a command request with the context params (either specified via the `ctl` shell tool or via Java API).
2. The dispatcher resolves the module. If the command is scoped to an object, it looks up the object's type and looks for an installed module by that name. If it is not an object scoped command, then the module name will have been specified already.
3. The dispatcher resolves the command. The dispatcher reads the module's metadata to understand how the command is declared. Overridden commands have extra metadata that tell the dispatcher what module contains it. If it is overridden, it will look in the implementing module for a command of that name.
4. The dispatcher loads the command handler file. At this time two properties are declared in the execution context: `module.dir` (the directory of the implementing module), `module.name` (the name of the implementing module).
5. The dispatcher executes the command handler.

### 2.3. Dispatching

Via dispatching, objects can call commands on themselves or commands on other objects. This is done via CTL's "controller" Java API which has access to the internal CTL dispatcher. The API is exposed as an Ant task and is simple to utilize via an `AntCommand` command-type.

The dispatcher sets various properties describing the sender. This information is accessible in the receiver's context. The table below shows them.

Property	Description
<code>dispatch.caller.command.name</code>	Name of the calling command

dispatch.caller.module.name	Module of calling command
dispatch.caller.cmd.line	Command line arg string of calling command
dispatch.caller.context.name	Sending object's name
dispatch.caller.context.type	Sending object's type
dispatch.caller.context.depot	Sending object's project

## 2.4. Instance data

You may recall there is a base type in CTL called "Managed-Entity". This base type establishes the idea of controllers, commands and properties. CTL provides a workspace for each object in a unique directory in the project depot accessible via the property, `#{entity.instance.dir}`.

Managed-Entity establishes a standard file location where this data can be maintained. You can get the path to this file via the property, `#{entity.properties.file}`. Both the `Get-Properties` and `Properties` commands update and print the content of the `#{entity.properties.file}` respectively. Since your commands consume and interpret the content of the property file it can actually be in any format you wish. That said, CTL uses the Java properties format and supplies its own data to an object's commands as such.

Take a look at the [entity.properties reference](#) for a complete listing of CTL supplied properties.

You can declare your own set of standard object attributes in your `type.xml` file using the [attribute-default](#) tags. After you build the type, the module will contain a file called `#{module.dir}/type.properties` containing a set of key/value pairs that correspond to your attribute-defaults. They will be in the form of:

```
entity.attribute.attributeName=typeDefaultValue
```

Your objects can use these attribute-defaults but also define their own values. The `#{module.dir}/type.properties` file is read after `#{entity.properties.file}`. Therefore, if your object declares values for the attributes the ones from the type are ignored.

You have some choice on how to maintain and distribute instance data for your object. The `Get-Properties` command can be overridden, too, so you can access data from any source you wish. If that is your intention, make sure all your modules derive from that one so they inherit your standard.

You can use Managed-Entity's implementation of `Get-Properties` command. Its implementation assumes you are maintaining all the object's `entity.properties` file on a web server and does simple GET HTTP requests to pull it whenever `Get-Properties` is invoked (typically via `Install`).

## 2.5. Calling a supertype command

Sometimes you want to add some additional steps to a command defined in a supertype. The name of your type's supertype is accessible via the property: `${type.supertype}` defined in your module's `type.properties` file.

In the example that follows you will see the "greeter" type is defined to override "poly" type, calling its "greet" command and then continuing with some of its own actions.

```
<type name="greeter">
  <description>
    Inherits poly's salute command but overrides
    poly's.
  </description>

  <supertype>
    <typereference name="poly"/>
  </supertype>
  <commands>
    <command name="greet" description="offers a salutation"
      command-type="AntCommand">
      <implementation>
        <controller>
          <execute>
            <context depot="${context.depot}"
              entityClass="${context.type}"
              entityName="${context.name}"/>
            <command name="salute" module="${type.supertype}"/>
          </execute>
        </controller>
        <!--
          ** greeters specific implementation
        -->
        <tstamp/> <!-- generates a timestamp, TSTAMP -->
        <!-- log the greeting to a file -->
        <echo message="said hello: ${TSTAMP}"
          file="${entity.instance.dir}/var/greet.log" append="true"/>
      </implementation>
    </command>
  </commands>
</type>
```

Some consider calling super an anti-pattern (see [\[Fowler\]](#)) recommending instead the use of a template method.

## 2.6. Template method pattern

CTL allows you to employ the template method design pattern. Wikipedia uses the following description:

### Template method

"A template method defines the program skeleton of an algorithm. The algorithm itself is made abstract, and the subclasses override the abstract methods to provide concrete behavior." ([link](#))

As the figure below shows, an AbstractClass declares a method which structures a series of other method calls in that class. The methods that are called can be overridden in sub classes.

For CTL, you don't need to make your type abstract. Carry out the steps below to implement the template method design pattern:

	Step	Where
1.	Define primitive operations	Supertype
2.	Define workflow to act as the template method. The workflow should invoke the primitive operations in the order desired.	Supertype
3.	Override one or more of the primitive operations.	Subtype

**Table 1: Steps**

After these steps are done and the modules built and deployed, you can call the workflow on the Subtype. You will find that the workflow command is inherited by the Supertype (and executed from within the Supertype's module) but the overridden commands of the Subtype are called.

## 3. Running commands

Running a command against an object is similar to running a static command. The `ctl` parameters change a bit to name the object you want to target. The general usage below shows the needed parameters. Note the `-t` and `-o` options.

General usage:

```
ctl -p project -t type -o object -c command [--[commandargs]]
```

The `-t` type option is similar to the `-m` module one but this lets CTL's dispatcher know how to locate the object and invoke the needed command.