AT Template Language Reference

| Product | DCG |
| --- | --- |
| Version | 3.0 |
| Author | Wei Yuan |
| Date | January 13, 2007 |

History

| Version | Date | Author | Description |
| --- | --- | --- | --- |
| 1.0 | January 13, 2007 | Wei Yuan | Try to be a reference manual for AT template language. |
| 1.1 | December 25, 2008 | Wei Yuan | Add a new language feature description. |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[Introduction 4](#_Toc218000424)

[Purpose 4](#_Toc218000425)

[Terms 4](#_Toc218000426)

[AT Sample 5](#_Toc218000427)

[AT Concepts 5](#_Toc218000428)

[Neutral Directives 6](#_Toc218000429)

[Comment 6](#_Toc218000430)

[Notation 6](#_Toc218000431)

[Sample 6](#_Toc218000432)

[Description 6](#_Toc218000433)

[Header Directives 6](#_Toc218000434)

[Reference 6](#_Toc218000435)

[Notation 6](#_Toc218000436)

[Sample 6](#_Toc218000437)

[Description 6](#_Toc218000438)

[Import 7](#_Toc218000439)

[Notation 7](#_Toc218000440)

[Sample 7](#_Toc218000441)

[Description 7](#_Toc218000442)

[Parameter 7](#_Toc218000443)

[Notation 7](#_Toc218000444)

[Sample 7](#_Toc218000445)

[Description 7](#_Toc218000446)

[Global Block 8](#_Toc218000447)

[Notation 8](#_Toc218000448)

[Sample 8](#_Toc218000449)

[Description 8](#_Toc218000450)

[Body Directives 8](#_Toc218000451)

[Execution 8](#_Toc218000452)

[Notation 8](#_Toc218000453)

[Sample 8](#_Toc218000454)

[Description 8](#_Toc218000455)

[Evaluation 9](#_Toc218000456)

[Notation 9](#_Toc218000457)

[Sample 9](#_Toc218000458)

[Description 9](#_Toc218000459)

[Multi-Line Evaluation 9](#_Toc218000460)

[Notation 9](#_Toc218000461)

[Sample 9](#_Toc218000462)

[Description 10](#_Toc218000463)

[Code Block 10](#_Toc218000464)

[Notation 10](#_Toc218000465)

[Sample 10](#_Toc218000466)

[Description 10](#_Toc218000467)

[Text Block 11](#_Toc218000468)

[Notation 11](#_Toc218000469)

[Sample 11](#_Toc218000470)

[Description 11](#_Toc218000471)

[Between Block 11](#_Toc218000472)

[Notation 11](#_Toc218000473)

[Sample 12](#_Toc218000474)

[Description 12](#_Toc218000475)

[Escape Character 12](#_Toc218000476)

[Notation 12](#_Toc218000477)

[Sample 13](#_Toc218000478)

[Description 13](#_Toc218000479)

[Line Ending Handling Character 13](#_Toc218000480)

[Notation 13](#_Toc218000481)

[Sample 13](#_Toc218000482)

[Description 13](#_Toc218000483)

[Output Block 13](#_Toc218000484)

[Notation 13](#_Toc218000485)

[Sample 14](#_Toc218000486)

[Description 14](#_Toc218000487)

[Section Definition 14](#_Toc218000488)

[Notation 14](#_Toc218000489)

[Sample 14](#_Toc218000490)

[Description 14](#_Toc218000491)

[Section Reference 15](#_Toc218000492)

[Notation 15](#_Toc218000493)

[Sample 15](#_Toc218000494)

[Description 15](#_Toc218000495)

# Introduction

## Purpose

This document is a reference for AT template language.

## Terms

| Term | Means |
| --- | --- |
| DCG | Dynamic Code Generator |
| AT | Character “@” based template language, it has many improved and unique template features as well as an easier way to extend itself. |
|  |  |
|  |  |

# AT Sample

@# This is a number listing template for demonstration.

@# You can render this template to see its output.

@param count: int

@code

for (int i = 1; i <= count; i++)

{

@text

@(i)

@end\_text

}

@end\_code

# AT Concepts

AT Syntax is constructed of directives. Directives are case sensitive and all directives are in lower case.

A template written with AT has two sections and two contexts. The two sections are:

* Header
* Body

The two contexts are:

* Static
* Dynamic

Contexts only exist in Body section. Static context means everything you write in this context is output as is, or what you write is a directive; Dynamic context means everything you write is .NET code, with ***@text*** directive being the only exception to this rule. You can check the following picture to clarify these concepts.

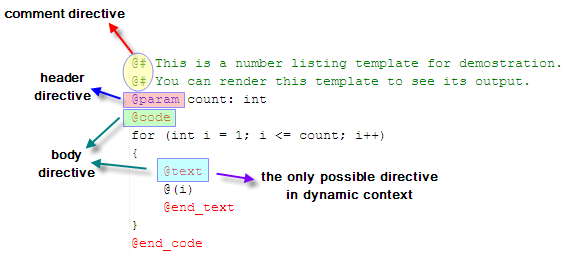


Image AT template parts

# Neutral Directives

## Comment

### Notation

@# <comment line>

### Sample

@# This line is a comment.

Hello there.

### Description

Comment directives can appear in both Header and Body. When appeared, the entire line should be comment.

# Header Directives

## Reference

### Notation

@reference <assembly path>

### Sample

@reference System.Windows.Forms.dll

@reference System.Data.dll

### Description

Use this directive to declare assemblies that your template references. You can reference assemblies in GAC, or you can reference your own assembly by specifying the assembly file.

The assembly path can be relative to template directory or be an absolute path.

By default, System.dll and Cavingdeep.Dcg.dll are automatically referenced so you have no need to explicitly declare them.

## Import

### Notation

@import <namespace>

or

@import <alias=(namespace | class)>

### Sample

@import System

@import MsgBox = System.Windows.Forms.MessageBox

### Description

Use this directive to import namespaces you used in your template. Its syntax is equivalent to C#’ *using*. You can also use alias just like you do in C#.

By default, namespace System is implicitly imported so you have no need to import it again.

## Parameter

### Notation

@param <name:type>

### Sample

@param name : string

@param age: int

@param account:decimal

### Description

Use this tag to declare template parameters. They are passed by the caller of template and they can be of any .NET type if template is running within the same AppDomain as caller or, they can be any serializable .NET type if template is running in a separate AppDomain.

## Global Block

### Notation

@global  
<Global C# code>  
@end\_global

### Sample

@global

int globalCount = 0;

int IncreaseCount()

{

return ++globalCount;

}

@end\_global

### Description

Use this directive to declare global members that could be accessed from within any part of the template. These global members are defined in C#.

# Body Directives

## Execution

### Notation

@! <C# code>

### Sample

@! int age = 0;

@! if (age > 0)

@! {

@! }

### Description

This directive is a line based dynamic code writing. It defines a line of C# code.

## Evaluation

### Notation

@(<C# expression>)

### Sample

@! int age = 1;

My age is @(age).

### Description

This directive executes the expression and returns its value. You use this directive to get the return value of an expression like a variable of a method call that returns some value.

## Multi-Line Evaluation

### Notation

@= <C# expression>

### Sample

Template

@global

string MethodThatReturnsMultipleLines()

{

return “Line 1\r\n” +

“Line 2”;

}

@end\_global

Bellow is an indented result:

@= MethodThatReturnsMultipleLines()

After result.

Output

Bellow is an indented result:

Line 1

Line 2

After result.

### Description

When a call to a method would return multiple lines and you want these lines being indented, you could use Multi-Line Evaluation directive. This is useful when you need to call another template that returns multiple lines in your template.

## Code Block

### Notation

@code  
<C# code>  
@end\_code

### Sample

@code

bool var1 = true;

if (var1)

{

//

}

else

{

//

}

@end\_code

### Description

Served as same purpose than [**@!**](#_Execution) directive, but this directive is not line based thus you don’t have to write it so many times.

The starting part of this directive puts you into dynamic context. That is, you can’t write static text nor use other directives in this context (with exception of [**@text**](#_Text_Block) directive). So what if you want to write some static text or use other directives? The answer is: You need to enter a static context, of course. By using @text directive you can just do that, entering a static context within a dynamic context. We will see this below.

When reaching the ending part of the directive, the dynamic context opened previously will close.

## Text Block

### Notation

@text  
<static text or directives>  
@end\_text

### Sample

Template

@code

for (int i = 1; i <= 3; i++)

{

@text

number @(i)

@end\_text

}

@end\_code

Output

number 1

number 2

number 3

### Description

This directive creates a static context inside a dynamic context, so you can write static text or use directives.

Notice that content of this directive must have the same indentation than the starting part of this directive. For example, if you start the @text directive with 4 spaces as indentation then its content should have 4 spaces as indentation too. The output of this directive is at its outside dynamic directive’s starting part position. For example, in the [sample](#_Sample) section above, numbers are outputted where “@code” appeared. If “@code” is indented then the numbers outputted will be indented too.

When reaching the ending part of the directive, the static context opened previously will close.

## Between Block

### Notation

<line end>@{  
<C# code>  
@}<rest of line>

### Sample

Template

int[] numbers = new int[] {@{

for (int i = 1; i <= 3; i++)

{

if (i+1 <= 3)

{

@text

@(i), @

@end\_text

}

else

{

@text

@(i)@

@end\_text

}

}

@}};

Output

int[] numbers = new int[] {1, 2, 3};

### Description

Use this directive if you want some dynamic output between some static texts. This will put you into dynamic context once entered.

You may observe the strange “@” character at the end of some lines. This is a [“Line Ending Handling Character”](#_Line_Ending_Handling) and we’ll see this later.

## Escape Character

### Notation

@@

### Sample

Template

What this outputs: @@(variable).

Output

What this outputs: @(variable).

### Description

When a conflicting situation may occur using “@”, the Escape character is always at your rescue.

## Line Ending Handling Character

### Notation

<line end>@

### Sample

Template

Line 1@

Line 2

Output

Line 1Line 2

### Description

Sometimes, you don’t want a line to break. In this kind of situations you can use this directive, just as you saw in [“Between Block”](#_Between_Block) sample.

## Output Block

### Notation

@output key  
<template content>  
@end\_output

### Sample

main output

@output alternateOutput

output in alternateOutput

@end\_output

### Description

Using this directive you can have more than one output in a template. The alternate outputs can be acquired using their key. Dcg offers an overloaded version of the **Render** method which supports multiple outputs. Please refer to the API document.

## Section Definition

### Notation

@section name(param1:type, param2:type, …)  
<section content>  
@end\_section

### Sample

<order>

@+ OrderDetail(orderObject)

</order>

@section OrderDetail(order: MyOrder)

<orderDetail>

@# Use order object here...

</orderDetail>

@end\_section

### Description

A section works like a template within template, it defines a piece of reusable content, you can reuse it by writing a [Section Reference](#_Section_Reference). A section can define zero or more parameters for flexible reuse, in this sense, it works like a C# method that you can call multiple times with different arguments.

When there is no parameter in the section, you can ignore the parenthesis.

## Section Reference

### Notation

@+ section\_name(param1, param2, …)

### Sample

<order>

@+ OrderDetail(orderObject)

</order>

@section OrderDetail(order: MyOrder)

<orderDetail>

@# Use order object here...

</orderDetail>

@end\_section

### Description

Section reference is used to refer to a section. It works in a way that you can consider it like a C# method call, it calls a section and returns the called section’s content.

When there is no argument in the calling section, you can ignore the parenthesis.