ETSI ES 203 790 V1.2.1 (2020-05)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

TTCN-3 Language Extensions: Object-Oriented Features

**ETSI Standard**

Reference

RES/MTS-203790-OOFv1.2.1

Keywords

language, TTCN-3

***ETSI***

650 Route des Lucioles

F-06921 Sophia Antipolis Cedex - FRANCE

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - NAF 742 C

Association à but non lucratif enregistrée à la

Sous-Préfecture de Grasse (06) N° 7803/88

***Important notice***

The present document can be downloaded from:  
<http://www.etsi.org/standards-search>

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format at [www.etsi.org/deliver](http://www.etsi.org/deliver).

Users of the present document should be aware that the document may be subject to revision or change of status. Information on the current status of this and other ETSI documents is available at <https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:  
<https://portal.etsi.org/People/CommiteeSupportStaff.aspx>

***Copyright Notification***

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.  
The content of the PDF version shall not be modified without the written authorization of ETSI.  
The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2020.

All rights reserved.

**DECT™**, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.  
**3GPP™**and **LTE™** are trademarks of ETSI registered for the benefit of its Members and  
of the 3GPP Organizational Partners.  
**oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and  
of the oneM2M Partners.  
**GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

#### 

#### 5.1.1.0 General

***Syntactical Structure***

[**public** | **private**]   
**type** [**external**] **class** [**@final** |**@abstract**]   
*Identifier* [**extends** *ClassType*]   
[*runsOnSpec*] [*systemSpec*] [*mtcSpec*]  
"{" {*ClassMember*} "}"   
[**finally** *StatementBlock*]

***Semantic Description***

A class is a type where the values are called objects. A class can declare fields (variables, constants, templates, timers, classes), methods and properties as its members. Each member name inside the class shall be unique, there is no overloading. The private and protected fields and methods are only accessible by the methods of the class, while the public members of the class can be accessed also from behaviour not defined in the class. The private members of the class can be accessed directly only by members of the class itself. All members which are neither private nor public are protected and can also be accessed by members of subclasses.

All fields may be declared without initializer, even const and template fields.

A class can extend another class. The extended class is called the superclass, while the extending class is called the subclass. The resulting type of a class definition is the set of object instances of the class itself and all instances of its direct or indirect subclasses. A subclass is a subtype of its direct and indirect superclasses and its object instances are type compatible with them. If a class does not explicitly extend another class type, it implicitly extends the root class type **object**. Thus, all classes are directly or indirectly extensions of the **object** class.

A class can have optional runs on, mtc and system clauses. This restricts the type of component context that can create objects of that class and all methods of this class. If a class does not have one of these clauses, it inherits it from its superclass, if the superclass has one. If the superclass has or inherits a runs on, mtc or system clause, the subclass may declare each of these clauses with a more specific component type than the one inherited. The function members of classes shall not have runs on, system or mtc classes but inherit them from their surrounding class or its superclasses.

***Restrictions***

1. Templates are not allowed for class types.
2. Passing of object references to the create operation of a component type or a function started on another component is not allowed.
3. No subtyping definition is allowed for class types via the normal subtype definition.
4. No local/global constants or module parameters of class type or containing class type fields or elements are allowed.
5. Class type cannot be the contained value of an anytype value.
6. The functions of a class shall not have a runs on, mtc or system clause.
7. The runs on type of a class shall be runs on compatible with the runs on type of the behaviour creating a class.
8. The runs on type of a class shall be runs on compatible with the runs on type of the superclass.
9. The mtc and system type of a class shall be mtc and system compatible with the mtc and system types of the superclass, respectively.

#### 5.1.1.2 Abstract classes

A class can be declared as @abstract. In that case, it is allowed that it also declares abstract member functions, abstract properties or properties with abstract getters or setters who shall be defined by all non-abstract subclasses. An abstract method function has no function body but can be called in all concrete instances of subclasses of the abstract class declaring it. Other members of the abstract class or its subclasses may use the abstract functions as if it was concrete where at runtime the concrete overriding definition will be used.

Abstract getters and setters have no body but the properties containing them can be referenced in all concrete instance of subclasses of the abstract class declaring them. Other members of the abstract class or its subclasses may reference abstract properties as if they were concrete. At runtime the concrete overriding definition will always be used.

NOTE 1: Abstract classes are only useful as superclasses of concrete classes.

***Restrictions***

1. Abstract classes cannot be explicitly instantiated.
2. If a class that is not declared abstract extends an abstract class, all methods, property getter and setters that have no implementation in the superclass shall be implemented in this class.

NOTE 2: Variables of an abstract class type can only contain references to instances of non-abstract subclasses.

#### 5.1.1.5 Constructors

***Syntactic Structure***

**create** "(" { *FormalParameter* , }\* ")"

**[ external** "(" { *FormalParameter* , }\* ")" ]  
[":" *ClassType* "(" { *ActualParameter* , }+ ")" ]   
[ *StatementBlock* ]

***Semantic Description***

A class may define a constructor called create.

If no constructor is defined inside a class body, an implicit default constructor is provided where the formal parameters of the constructor are the parameters of the (implicit or explicit) constructor of the direct superclass and one additional formal in parameter for each declared **var** field of the class itself and also all **const** or **template** fields or automatic properties with no initializer in their order of declaration with the same type as in the declaration.

The constructor is invoked on a type reference to the class and the result of this invocation is a new instance object of the constructor's specific class. If a class is extending another class with a constructor with at least one parameter without default, that constructor shall be invoked by adding a super-constructor clause to the constructor declaration. The super-constructor clause consist of a reference to the class being extended and an actual parameter list. An implicit constructor will automatically pass the required actual parameters to the constructor of its superclass.

In the constructor, it is allowed to refer to the object being constructed as this to reference the fields of the object to be created in case that the names of the formal parameters clash with the names of those fields. They are explicitly allowed to have the same names as class members.

When an object is created via the invocation of a constructor, the fields of each class body in the class hierarchy that have initializers are initialized before the execution of that class body’s constructor body. The fields of a superclass that have initializers are initialized before the fields of the subclass. Also, the constructor of the superclass is executed before the constructor body of the subclass. Thus, it is ensured that all initialization of the superclass hierarchy as well as local fields with initializers is finished before the execution of a constructor body.

Since the members of a class body can appear in any order and forward references are allowed between them, a field with an initializer which is referenced by the initializer of another field, is initialized first.

As the underlying external constructor of external classes might need additional parameters, these can be provided via the additional external formal parameter list. If no internal constructor needs to be defined, the constructor may be defined without external formal parameter list and no body. In that case, the formal parameter list defines the formal parameters passed to the external constructor.

***Restrictions***

1. All formal parameters of the constructor shall be **in** parameters.
2. The constructor body shall not assign anything to variables that are not local to the constructor body or accessible fields of the class the constructor belongs to.
3. The constructor body shall not use blocking operations.
4. The initialization of a member field shall not invoke any member function in the object being initialized.
5. The constructor body shall not invoke any member function in the object being initialized.
6. A member constant or template shall be initialized exactly once, either by its initialization part or by at most one constructor body.
7. Direct or indirect cyclic initialization is not allowed. That is the initializer of a field shall not use the same field directly or indirectly.
8. The initializer of a field shall not use a field that does not have an initializer.

EXAMPLE 1:

**type** **class** MyClass {

**var** **integer** a;

**const** **float** b;

const float c := 7;

template float myTemplate := ?;

// implicit constructor:

// only using variable fields and non-variable fields with no initializer

//**create**(**integer** a, **float** b) { // no parameter for c and myTemplate

// **this**.a := a;

// **this**.b := b

//}

}

**type** **class** MyClass2 extends MyClass {

**template** **integer** t;

// explicit constructor

**create**(**template** **integer** t) : MyClass(2, 0.5) {

this.t := t;

}

}

type class MyClass3 extends MyClass {

var float f;

// implicit constructor:

// create(integer a, float b, float f) : MyClass(a, b) {

// this.f := f;

// }

}

EXAMPLE 2:

For each initialization statement it is marked with its initialization order in the comment.

**type** **class** MySuperClass {

**var** **integer** a := 5; // 1

**const** **float** b;

**create**(**integer** a, **float** b) {

**this**.a := a; // 3

**this**.b := b; // 4

}

}

**type** **class** MySubClass **extends** MySuperClass {

**var** **template** **integer** t := ?; // 2

**create**(**template** **integer** t) : MySuperClass(2, 0.5) {

**this**.t := t; // 5

}

}

#### 5.1.1.13 Properties

***Syntactic Structure***

@**property** [ *TemplateModifier* ] { ( **@abstract** | **@final** | **@deterministic )**} *Type Identifier*

[ ":=" *TemplateBody* ]

[ **"{"**

[ { ( **@abstract** | **@final** | **@deterministic** ) }

**@get** [

("=>" *TemplateBody* [";"]) |

("{" *StatementBlock* "}" )

]

]

[ { ( **@abstract** | **@final** | **@deterministic ) }**

**@set** [

("=>" *Assignment* [";"]) |

("{" *StatementBlock* "}" )

]

]

"}" ]

***Semantic Description***

A class property is a class member which is referenced like a record field for reading and writing with the dotted notation, but implemented via getter and setter functions that are provided in the definition of the property (allowing value checking/normalization/conversion when setting a value and on-the-fly computation when getting the value).

Properties are in many regards similar to member functions, they can be declared with modifiers and it is allowed to override them in subclasses.

Modifiers can occur either on the property level or in a getter and setter declaration. Modifiers declared on a property level are valid for both the getter and setter.

While most properties are declared with a property body containing either a getter or setter function, it is allowed to declared a property without a body. Such a property is called automatic property. Automatic properties are similar to member variables, they are always generated with an associatied anonymous member variable and a getter that returns this variable and a setter that assigns a value to this variable. While they are similar to member variables, there are two important differences: automatic properties can be public and it is possible to override them in subclasses. If a class contains an implicit default constructor, this constructor contains a formal parameter for each anonymous member variable associated with an automatic property.

All getters and setters shall have a body unless they are declared abstract. A body of a getter can be specified in two distinct forms: either in a simple form that contains a single *TemplateBody* or in an extended form that consist of a statement block. In case of the simple form, referencing the property on the right hand side of an assignment will return the *TemplateBody* referenced in the getter declaration. When the extended form is referenced on the right hand side of an assignment, the statement block of the getter function will be executed and the value from the **return** clause of this block will be returned.

A body of a setter can be specified in two distinct forms: either in a simple form that contains a single assignment or in an extended form that consist of a statement block. The setter is executed when the property is referenced on the left hand side of an assignment. Both forms of the setter may reference a special variable **value**. This variable works as an **in** formal parameter of the setter function. It is of the same type as the property itself and when the setter is invoked, the value from the left hand side of the assignment where the property was referenced is passed into it as an actual parameter according to the rules specified in the section 5.4.2 of [1].

Properties that contain a setter (including automatic properties that have an implicit setter) may be optionally declared with an initial value. The initial value follows the identifier of the property and is preceded by an assign symbol. The initial value is automatically passed to the setter when an instance of the defining class is created. This automatic invocation takes place after execution of a constructor of the parent class and before execution of the constructor of the defining class. Properties are automatically initialized in the declaration order.

Abstract properties shall always contain a property body. Getters and setters declared inside an abstract property shall have no body. If the **@abstract** modifier is attached to a getter or setter, the getter or setter shall have no body.

***Restrictions***

1. With the exception of a special case of automatic properties, a property shall always have a getter or a setter or both of them. Empty property body is not allowed.
2. The *TemplateBody* in the simplified form of the getter function and in the return clause of the common form of the getter function shall be compatible with the property type according to the rules specified in the section 6.3 of [1]. If the property has no *TemplateModifier*, the *TemplateBody* shall contain a value. Otherwise, it might contain a template that fulfils the restrictions set by the *TemplateModifier* that are specified in the section 15.8 of [1].
3. The **value** variable of the setter function has the same template modifiers as the property itself.
4. When passing a value to the value variable of the setter function, the rules specified in the section 5.4.2 of [1] shall apply.
5. If a property or its getter or setter contain the **@deterministic** modifier, rules for derministic functions specified in the section 16.1.0 of [1] shall apply to the body of the concerned getter or setter.
6. When overriding a property, it is allowed to add a getter or setter even if it was not present in the parent class. Such a getter or setter is available only in the overriding class and its subclasses.
7. When overriding a property, if a getter or setter is present in the parent class but not in the overriding class, then the getter or setter from the parent class is valid for the overriding class as well.
8. An error shall be produced when a property that has no getter is referenced on the right hand side of an assignment.
9. An error shall be produced when a property that has no setter is referenced on the left hand side of an assignement.
10. An error shall be produced if execution of the extended form of a getter is terminated by reaching the end of the statement block without execuring a **return** statement or a statement that terminates component execution (such as **stop** or **testcase.stop**).
11. Initial value of a property shall be compatible with the property type. If the property has no *TemplateModifier*, the initial value shall resolve into a value. Otherwise, it might contain a template that fulfils the restrictions set by the *TemplateModifier* of the property. These restrictions are specified in the section 15.8 of [1].

EXAMPLE:

**type class** Rectangle {

**private var int** heightVal;

**public @property int** width; // automatic property width

**public @property int** height { // property with a getter and setter

**@get** => heightVal; // simple form of a getter

**@set** { // extended form of a setter

if (**value** > 0) { // simple data integrity check

heightVal := **value**;

}

}

}

**public @property int** perimeter {

**@get** { // extended form of a getter

**return** 2 \* (width + height);

}

}

}

…

**var** Rectangle v\_rect := Rectangle.**create**(10, 9); // instantiation using an implicit constructor

v\_rect.width := 16; // change the width to 16

**log** (v\_rect.perimeter); // prints 144 to the log

v\_rect.perimeter := 100; // causes an error as the referenced property has no setter