ETSI ES 203 790 V1.1.1 (2019-01)

Methods for Testing and Specification (MTS);

The Testing and Test Control Notation version 3;

TTCN-3 Language Extensions: Object-Oriented Features

**ETSI Standard**

Reference

DES/MTS-203790-00F\_ed111

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 only prevailing document is the print of the Portable Document Format (PDF) version kept on a specific network drive within ETSI Secretariat.

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 2019.

All rights reserved.

**DECT**TM, **PLUGTESTS**TM, **UMTS**TM and the ETSI logo are trademarks of ETSI registered for the benefit of its Members.
**3GPP**TM and **LTE**TM 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 Package Concepts for the Core Language

## 5.0 General

This package defines objec-oriented features for TTCN-3, i.e. it extends the TTCN-3 core language (ETSI ES 201 873‑1 [1]) with well-known concepts from object-oriented programming and modelling languages. This package realizes the following concepts:

* classes (i.e. class definition, scope rules, abstract and external classes, refinement, constructors, destructors, methods, visibility, and built-in classes);
* objects (i.e. ownership, object references, select class-statement, dynamic class discrimitation, and casting); and
* exception handling (i.e. ability to define exception handling for functions, external functions, altsteps, and test cases).

## 5.1 Classes and Objects

### 5.1.0 General

This clause introduces the concepts of class types and their values, called objects as well as the operations allowed to be applied to these objects.

### 5.1.1 Classes

#### 5.1.1.0 General

***Syntactical Structure***

[**public** | **private**]
**type** [**external**] **class** [**@final** |**@abstract** | @trait]
*Identifier* [**extends** *Identifier* {"," *Identifier*}]
[*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, ports, timers) and methods 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.

A class can be declared with the @trait modifier. Such a class is called a trait class. Other classes are called normal classes. A trait class is an abstract class and can not be instantiated. It also shall not declare a constructor and the declared member fields shall not have initializers.

A normal class can extend at most one other normal class and also any number of trait classes. The extended normal class is called the superclass, the extended trait classes are called the supertraits, while the extending class is called the subclass of all the classes it extends. Trait classes can only extend trait classes but not normal classes. 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 supertraits 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 inherits all members of its superclass and its supertraits that it does not override in its own class body and which have not been overridden in the class body of one of the extended classes. A non-private non-abstract member from the superclass or one of the supertraits can always be accessed inside the class body by using the dotted notation on the keyword **super**. Non overridden non-private members can be accessed without any dotted notation before the member name.

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 and the supertraits.
9. The mtc and system type of a class shall be mtc and system compatible with the mtc and system types of the superclass and the supertraits, respectively.
10. If a class extends multilple classes, the classes being extended shall not contain fields with the same name defined in different class bodies.
11. If a class extends multiple classes, the classes being extended shall not contain methods with the same name defined in different class bodies unless one of methods overrides directly or indirectly all the other methods with the same name.

NOTE: Please note that this also means, that while adding a new overriding function to an existing normal or trait class is semantically valid in its own context, it can be a non-backward compatible change for subclasses and sub traits, if they also extend an other (possibly independent) trait that also overrides the same function.
12. Class extension shall not contain cycles such that a class directly or indirectly extends itself.
13. Reference to a class shall not occur more than once in the list of classes being extended.
14. Fields declared in trait classes shall not have initializers.
15. Trait classes shall not define a constructor and shall not define a finally block.

EXAMPLE 1:

external function newGlobalId() return charstring;

type class @trait Identifiable {
  private var charstring id;
  public function setId(charstring id) {
    this.id := id;
  }

  public function getId() return charstring {
    return id;
  }
}

type class MyIdentifiableClass extends Identifiable {
  create() {
    setId(newGlobalId());
  }
}

var Identifiable v\_idObj := MyIdentifiableClass.create();
var charstring v\_id := v\_idObj.getId();

EXAMPLE 2: parallel inheritance

type class @trait A {
  function @abstract f();
}

type class @trait B {
  function f() { … }
}

type class C extends A, B {
  // illegal, as it inherits A.f() and B.f() neither of which overrides the other
}

type class @trait B2 extends A {

 function f() { … } // overrides A.f()

}

type class C2 extends A, B2 { // legal, as it inherits only B2.f()
  function f() { ... } // overrides B2.f()
}

type class C3 extends A {
  function f() { ... } // overrides A.f()
}

type class D extends C2, C3 {
  // illegal, as it inherits C2.f() and C3.f()
}

type class E extends A, C2 {

 // legal, it does not inherit A.f() as this is overriden by C2.f()

}

#### 5.1.1.1 Scope rules

Class constitutes a scope unit. For the uniqueness of identifiers, the rules specified in the clause 5.2.2 of ETSI ES 201 873-1 [1] apply with the following exceptions:

1. Identifiers from the higher scope can be reused for member declarations. A reference to a reused identifier without a prefix occurring inside a class scope shall be resolved as a reference to the class member. In order to refer to the declaration on the higher scope, the identifier shall be preceded with a module name and a dot (".").
2. Identifiers of member declarations can be reused inside methods for formal parameter and local declarations. A reference to a reused identifier without a prefix occurring inside a class method shall be resolved as a reference to the formal parameter or local declaration. In order to refer to the member declaration, the identifier shall be preceded with the this keyword and a dot.
3. Reusing identifiers of members of the component type specified in the runs on clause of the class for members and inside methods for formal parameters and local declarations is not allowed.

EXAMPLE:

**module** ClassModule {

 **const** **integer** a := 1;

 **type class** MyClass() {

 **const** **integer** a := 2;

 **function** doSomething(**integer** a := 3) {

 log(a); // logs 3 (for the default value)

 log(**this**.a); // logs 2

 log(ClassModule.a); // logs 1

 }

 **function** doSomethingElse () {

 log(a); // logs 2

 log(**this**.a); // also logs 2

 log(ClassModule.a); // logs 1

 }

 }

}

#### 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 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.

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 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.

#### 5.1.1.3 External classes

A class may also be declared as external. In that case, all members shall be external functions without a function body. It is allowed to omit the external keyword from these function declarations. When instantiating an external class, the object being created is provided by the platform adapter and the method calls to the object are delegated via the platform adapter to the corresponding method of the external object.

NOTE 1: External classes are a way to use object-oriented library functionality to TTCN-3 while still remaining abstract and independent of actual implementation. Libraries for common constructs like stacks, collections, tables can be defined or automatic import mechanisms could be provided.

If an object of an external class is instantiated, it implicitly creates an external object and the internal object has a handle to the external one. The reference to the external object is called a handle. When an external method is invoked on the internal object, the call is delegated to the handle.

NOTE 2: External objects are possibly shared between different parts of the test system. Therefore, racing conditions and deadlocks have to be avoided by the external implementation.

An internal class can extend an external class and add internal behaviour, but also additional external functions, which have to be declared external explicitly. Such a class is conceptually still an external class and each instance has a handle to an external object.

***Restrictions***

1. External classes shall not contain fields or functions with a body.
2. External classes shall not be derived from non-external classes other than object.
3. If an internal class defines an external function, it shall be derived from an external class either directly or indirectly.

EXAMPLE:

external type class Stack {

 function push(integer v);

 function pop() return integer;

 function isEmpty() return boolean;

}

#### 5.1.1.4 Final Classes

If a class shall not be subclassed, it may be declared as @final. Final classes cannot be abstract.

#### 5.1.1.5 Constructors

***Syntactic Structure***

**create** "(" { *FormalParameter* , }\* ")"
[":" *SuperClass* "(" { *ActualParameter* , }+ ")" ]
*StatementBlock*

***Semantic Description***

A class can define a constructor called create. If no constructor is defined, a default constructor is implicitly provided where the formal parameters of the constructor are the parameters of the (implicit or explicit) constructor of the direct superclass and one formal in parameter for each declared member field of the class itself in their order of declaration with equivalent type. 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 an explicit constructor, that constructor shall be invoked by adding a super-constructor clause with an actual parameter list to the constructor declaration. 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.

EXAMPLE:

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

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

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

 // implicit constructor:

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

 // **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;

 }

}

#### 5.1.1.6 Destructors

***Syntactic Structure***

**finally** *StatementBlock*

***Semantic Description***

A destructor may be provided using a finally declaration following the class body. This destructor will be invoked automatically at the latest before the system deallocates an object instance (which is tool specific and out of the scope of the present document) or when the owning component is terminates. The *StatementBlock* has access to all members accessible to the class. The *StatementBlock* is semantically a function body of a function without return clause.

When deallocating the object instance, the destructor of the associated class is invoked first, followed by the destructor of all parent classes in the reverse order of superclass hierarchy.

#### 5.1.1.7 Methods

A method is a function defined inside the class body. It has the same properties and restrictions as any normal function, but it is invoked in an object which can be referred to by the this object reference. A method invocation can access the class's own fields and also the inherited protected fields and methods of its superclasses.

A method inherited from a superclass can be overridden by the subclass by redefining a function of the same name and with the same formal parameter list. When a method is called in an object, the version of the most specific class of the super class hierarchy of the concrete class that defines the method in its body will be invoked. The overridden method can be invoked from the overriding class by using the keyword super as the object reference of the invocation. If a method shall not be overridden by any subclass, it can be declared as @final.

Public methods, if not overridden by the subclass, are inherited from the superclasses. If a public method is declared in a class, it can be invoked also in all objects of its direct or indirect subclasses.

If a public method is overridden, the overriding method shall have the same formal parameters in the same order as the overridden method. Public methods shall be overridden only by public methods. Protected methods may be overridden by public or protected methods.

The return type of an overriding function shall be the same as the return type of the overridden function with the same template restrictions and modifiers.

Methods shall have no runs on, system or mtc clause directly attached to them. However, they inherit these clauses from their surrounding class.

#### 5.1.1.8 Method invocation

***Syntactical Structure***

[ObjectInstance "."] Identifier "(" FunctionActualParList ")"

A method invocation is a function call associated with a certain object defined in the class of that object.

Methods are invoked using the dotted notation on an object reference. Inside the scope of a class, methods of the same class or any visible inherited methods can be invoked without the *ObjectInstance* prefix if the object the method shall be invoked in is the same object as the one invoking it. The usual restrictions on actual parameters, as well as runs on, mtc and system types apply also on method invocations. All other restrictions that apply to called functions also apply to method invocation.

#### 5.1.1.9 Visibility

Fields can be declared as private or protected. Methods can be declared as private, public or protected. If no visibility is given then the default modifier protected is assumed.

Private member functions are not visible and can be present in multiple classes of the same hierarchy with different parameter lists and return values.

Public member functions can be called from any behaviour running on the object's owner component.

***Restrictions***

1. A field of any visibility cannot be overridden by a subclass.
2. A public member function can only be overridden by another public member function.
3. Private members can only be accessed directly from inside their surrounding class's scope.

#### 5.1.1.10 Built-in classes

The abstract special built-in class called object is the superclass for all classes that do not explicitly extend another class.

The pseudo definition of that class is:

type class @abstract @builtin object {

 // This function will return a tool-specific descriptive string by default

 // but can be overridden by subclasses
   public function toString() return universal charstring;
}

NOTE: The @builtin is only added for illustrative purposes and not part of the TTCN-3 language.

# A.3 Additional TTCN-3 syntax BNF productions

This clause includes all additional BNF productions that needed to define the syntax introduced by this package. All rules start with the digits "0330".

Additional BNF rules related to clause A.1.6.1.1 Type definitions

033001. ClassDef ::= [ [ExtKeyword](#ExtKeyword) ] [ClassKeyword](#ClassKeyword)

 [ [FinalModifier](#FinalModifier) | [AbstractModifier](#AbstractModifier) | TraitModifier ]

 [Identifier](#Identifier) [ [ExtendsKeyword](#ExtendsKeyword) [Type](#Type)List] [ [RunsOnSpec](#RunsOnSpec) ] [ [MtcSpec](#MtcSpec) ] [ [SystemSpec](#SystemSpec) ]

 "{" [ClassMemberList](#ClassMemberList) "}"

 [ [FinallyKeyword](#FinallyKeyword) [BasicStatementBlock](#BasicStatementBlock) ]

033002. ClassKeyword ::= "class"

033003. ThisOp ::= "this"

033004. SuperOp ::= "super"

033005. FinalModifier ::= "@final"

033006. AbstractModifier ::= "@abstract"

033007. FinallyKeyword ::= "finally"

033008. ObjectType ::= "object"

033009. ClassMemberList ::= { [ClassMember](#ClassMember) [ [WithStatement](#WithStatement) ] [ [SemiColon](#SemiColon) ] }

033010. ClassMember ::= [ [MemberVisibility](#MemberVisibility) ]

 ( [VarInstance](#VarInstance) |

 [TimerInstance](#TimerInstance) |

 [ConstDef](#ConstDef) |

 [TemplateDef](#TemplateDef) |

 [ClassFunctionDef](#ClassFunctionDef) |

 [ConstructorDef](#ConstructorDef) )

033011. MemberVisibility ::= "public" | "private"

033012. ClassFunctionDef ::= [ [ExtKeyword](#ExtKeyword) ] [FunctionKeyword](#FunctionKeyword)

 [ [FinalModifier](#FinalModifier) | [AbstractModifier](#AbstractModifier) ] [ [DeterministicModifier](#DeterministicModifier) ]

 [Identifier](#Identifier) "(" [ [FunctionFormalParList](#FunctionFormalParList) ] ")" [ [ReturnType](#ReturnType) ]

 [ [StatementBlock](#StatementBlock) ]

033013. ConstructorDef ::= [CreateKeyword](#CreateKeyword)

 "(" [FunctionFormalParList](#FunctionFormalParList) ")" [ ":" [FunctionInstance](#FunctionInstance) ]

 [ [StatementBlock](#StatementBlock) ]

033014. TraitModifier ::= "@trait"

003015. TypeList ::= Type { "," Type }

Additional BNF rules related to clause A.1.6.1.4 Function definitions

033014. BasicStatementBlock ::= "{" [ [FunctionDefList](#FunctionDefList) ] [ [FunctionStatementList](#FunctionStatementList) ] "}"

033015. CatchBlocks ::= [CatchBlock](#CatchBlock) { [CatchBlock](#CatchBlock) }

033016. CatchBlock ::= [CatchOpKeyword](#CatchOpKeyword) "(" [Type](#Type) [Identifier](#Identifier) ")" [BasicStatementBlock](#BasicStatementBlock)

033017. FinallyBlock ::= [FinallyKeyword](#FinallyKeyword) [BasicStatementBlock](#BasicStatementBlock)

033018. ObjectInstance ::= ( [ExtendedIdentifier](#ExtendedIdentifier) | [FunctionInstance](#FunctionInstance) ) [ [ExtendedFieldReference](#ExtendedFieldReference) ]

Additional BNF rules related to clause A.1.6.8.2 Behaviour statements

033019. RaiseExceptionStatement ::= [RaiseKeyword](#RaiseKeyword) [TemplateInstance](#TemplateInstance)

/\* STATIC SEMANTICS - The TemplateInstance shall evaluate to an explicit value. \*/

# History

|  |
| --- |
| **Document history** |
| V1.1.1 | October 2018 | Membership Approval Procedure MV 20181225: 2018-10-26 to 2018-12-25 |
| V1.1.1 | January 2019 | Publication |
|  |  |  |
|  |  |  |
|  |  |  |