ETSI ES 203 790 **VX.X.X** (**2022**-**XX**)

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-203790v131

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 - APE 7112B

Association à but non lucratif enregistrée à la

Sous-Préfecture de Grasse (06) N° w061004871

***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>

***Notice of disclaimer & limitation of liability***

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or

other professional standard and applicable regulations.

No recommendation as to products and services or vendors is made or should be implied.

No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.

In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

***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 2021.

All rights reserved.

Contents

Intellectual Property Rights 5

Foreword 5

Modal verbs terminology 5

1 Scope 6

2 References 6

2.1 Normative references 6

2.2 Informative references 6

3 Definition of terms, symbols and abbreviations 7

3.1 Terms 7

3.2 Symbols 7

3.3 Abbreviations 7

4 Package conformance and compatibility 7

5 Package Concepts for the Core Language 8

5.0 General 8

5.1 Classes and Objects 8

5.1.0 General 8

5.1.1 Classes 8

5.1.1.0 General 8

5.1.1.1 Scope rules 10

5.1.1.2 Abstract classes 11

5.1.1.3 External classes 11

5.1.1.4 Final Classes 12

5.1.1.5 Constructors 12

5.1.1.6 Constructor invocation 14

5.1.1.7 Destructors 14

5.1.1.8 Methods 15

5.1.1.9 Method invocation 15

5.1.1.10 Visibility 15

5.1.1.11 Built-in classes 16

5.1.1.12 Properties 16

5.1.2 Objects 18

5.1.2.0 General 18

5.1.2.1 Ownership 19

5.1.2.2 Object References 19

5.1.2.3 Null reference 19

5.1.2.4 Select class-statement 19

5.1.2.5 Of-operator (Dynamic Class Discrimination) 20

5.1.2.6 Casting 20

5.1.2.7 Comparison 20

5.1.2.8 Object Templates 21

5.1.3 Extension to ETSI ES 201 873-1, clause 7.1.8 (Presence checking operators) 22

5.2 Exception handling 23

5.2.0 General 23

5.2.1 Extension to ETSI ES 201 873-1, clause 16.1.0 (Functions) 23

5.2.2 Extension to ETSI ES 201 873-1, clause 16.1.3 (External Functions) 24

5.2.3 Extension to ETSI ES 201 873-1, clause 16.1.4 (Invoking functions from specific places) 24

5.2.4 Extension to ETSI ES 201 873-1, clause 16.2 (Altsteps) 24

5.2.5 Extension to ETSI ES 201 873-1, clause 16.3 (Test cases) 25

5.2.6 Extension to ETSI ES 201 873-1, clause 18 (Overview of program statements and operations) 25

5.2.7 Extension to ETSI ES 201 873-1, clause 19 (Basic program statements) 27

6 TRI Extensions for the Package 30

6.1 Extensions to clause 5.3 of ETSI ES 201 873-5 Data interface 30

6.2 Extensions to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations 31

6.3 Extensions to clause 6 of ETSI ES 201 873-5 JavaTM language mapping 33

6.4 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping 34

6.5 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping 35

6.6 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping 36

7 TCI Extensions for the Package 37

7.1 Extensions to clause 7.2.2.1 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values 37

7.2 Extensions to clause 7.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values 37

7.3 Extensions to clause 7.2.2.2.0 of ETSI ES 201 873-6 Basic rules 38

7.4 Extensions to clause 7.2.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 values 39

7.5 Extensions to clause 7.3.4.1 of ETSI ES 201 873-6 Abstract TCI-TL provided 40

7.6 Extensions to clause 8 of ETSI ES 201 873-6 JavaTM language mapping 44

7.7 Extensions to clause 9 of ETSI ES 201 873-6 ANSI C language mapping 46

7.8 Extensions to clause 10 of ETSI ES 201 873-6 C++ language mapping 48

7.9 Extensions to clause 11 of ETSI ES 201 873-6 W3C XML mapping 50

7.10 Extensions to clause 12 of ETSI ES 201 873-6 C# language mapping 52

8 XTRI Extensions for the Package (optional) 54

8.1 Changes to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations 54

8.2 Extensions to clause 6 of ETSI ES 201 873-5 JavaTM language mapping 56

8.3 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping 56

8.4 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping 57

8.5 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping 57

Annex A (normative): BNF and static semantics 58

A.1 Extensions to TTCN-3 terminals 58

A.2 Modified TTCN-3 syntax BNF productions 59

A.3 Additional TTCN-3 syntax BNF productions 60

Annex B (normative): Standard Collections 62

B.1 The TTCN3\_standard\_collections module 62

B.1.0 General 62

B.1.1 The Collection class 63

B.1.2 The List class 63

B.1.3 The LinkedList class 63

B.1.4 The Queue class 64

B.1.5 The PriorityQueue class 64

B.1.6 The Stack class 65

B.1.7 The RingBuffer class 65

B.1.8 The HashMap class 66

B.1.9 The Set class 67

B.1.10 The Exception class 67

B.1.11 The Iterator class 67

History 68

# Intellectual Property Rights

Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the ETSI Web server (<https://ipr.etsi.org/>).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

**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.

# Foreword

This ETSI Standard (ES) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

**The use of underline (additional text) and strike through (deleted text) highlights the differences between base document and extended documents.**

The present document relates to the multi-part standard ETSI ES 201 873 covering the Testing and Test Control Notation version 3, as identified in ETSI ES 201 873-1 [1].

# Modal verbs terminology

In the present document "**shall**", "**shall not**", "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the [ETSI Drafting Rules](https://portal.etsi.org/Services/editHelp!/Howtostart/ETSIDraftingRules.aspx) (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

# 1 Scope

The present document defines the support for object-oriented features in TTCN-3. TTCN-3 can be used for the specification of all types of reactive system tests over a variety of communication ports. Typical areas of application are protocol testing (including mobile and Internet protocols), service testing (including supplementary services), module testing, testing of OMG CORBA based platforms, APIs, etc. TTCN-3 is not restricted to conformance testing and can be used for many other kinds of testing including interoperability, robustness, regression, system and integration testing. The specification of test suites for physical layer protocols is outside the scope of the present document.

TTCN-3 packages are intended to define additional TTCN-3 concepts, which are not mandatory as concepts in the TTCN-3 core language, but which are optional as part of a package which is suited for dedicated applications and/or usages of TTCN-3.

While the design of TTCN-3 package has taken into account the consistency of a combined usage of the core language with a number of packages, the concrete usages of and guidelines for this package in combination with other packages is outside the scope of the present document.

# 2 References

## 2.1 Normative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

Referenced documents which are not found to be publicly available in the expected location might be found at <https://docbox.etsi.org/Reference/>.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are necessary for the application of the present document.

[1] ETSI ES 201 873-1: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language".

[2] ETSI ES 201 873-4: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 4: TTCN-3 Operational Semantics".

[3] ETSI ES 201 873-5: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 5: TTCN-3 Runtime Interface (TRI)".

[4] ETSI ES 201 873-6: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 6: TTCN-3 Control Interface (TCI)".

## 2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non‑specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long term validity.

The following referenced documents are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] ETSI ES 201 873-7: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 7: Using ASN.1 with TTCN-3".

[i.2] ETSI ES 201 873-8: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 8: The IDL to TTCN-3 Mapping".

[i.3] ETSI ES 201 873-9: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 9: Using XML schema with TTCN-3".

[i.4] ETSI ES 201 873-10: "Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 10: TTCN-3 Documentation Comment Specification".

# 3 Definition of terms, symbols and abbreviations

## 3.1 Terms

For the purposes of the present document, the terms given in ETSI ES 201 873-1 [1], ETSI ES 201 873‑4 [2], ETSI ES 201 873-5 [3] and ETSI ES 201 873-6 [4] apply.

## 3.2 Symbols

Void.

## 3.3 Abbreviations

For the purposes of the present document, the abbreviations given in ETSI ES 201 873-1 [1], ETSI ES 201 873‑4 [2], ETSI ES 201 873-5 [3] and ETSI ES 201 873-6 [4] apply.

# 4 Package conformance and compatibility

The package presented in the present document is identified by the package tag:

"TTCN-3:2018 Object-Oriented features" - to be used with modules complying with the present document*.*

For an implementation claiming to conform to this package version, all features specified in the present document shall be implemented consistently with the requirements given in the present document and in ETSI ES 201 873‑1 [1] and ETSI ES 201 873‑4 [2].

The package presented in the present document is compatible to:

* ETSI ES 201 873-1 [1], version 4.10.1;
* ETSI ES 201 873-4 [2], version 4.6.1;
* ETSI ES 201 873-5 [3], version 4.8.1;
* ETSI ES 201 873-6 [4], version 4.9.1;
* ETSI ES 201 873-7 [i.1];
* ETSI ES 201 873-8 [i.2];
* ETSI ES 201 873-9 [i.3];
* ETSI ES 201 873-10 [i.4].

If later versions of those parts are available and should be used instead, the compatibility to the package presented in the present document has to be checked individually.

# 5 Package Concepts for the Core Language

## 5.0 General

This package defines object-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 discrimination 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** *ClassType* {"," *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, 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 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 only declare methods without function bodies as members and no constructor.

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. A non-private non-abstract member from the superclass 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.

Classes can be used as field or element types of structured types.

***Restrictions***

1. Void.
2. Passing of object references and structured types containing fields or elements of class type 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 structured types containing fields or elements of class type are allowed.
5. Class type cannot be the contained value of an any type 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. Class extension shall not contain cycles such that a class directly or indirectly extends itself.
11. Reference to a class shall not occur more than once in the list of classes being extended.
12. Neither fields not non-abstract methods shall be declared in trait classes.
13. Trait classes shall not define a constructor and shall not define a finally block.
14. A class shall extend at most one normal class.
15. If a structured type contains a field of a class type, this type is not seen as a data type and its values cannot be used for encoding or decoding, sending or receiving and neither used as an actual parameter (or part thereof) to a function started on another component.

***Examples***

EXAMPLE 1:

**external function** newGlobalId() **return charstring**;

**type class @trait** Identifiable {

**public function @abstract** setId(**charstring** id);

**public function @abstract** getId() **return charstring**;

}

**type class** MyIdentifiableClass **extends** Identifiable {

**create**() {

setId(newGlobalId());

}

**var charstring** id;

**public function** setId(**charstring** id) { **this**.id := id }

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

}

**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 @abstract** f();

}

**type class** C **extends** A, B {

// legal, as it inherits A.f() and B.f() and they have the same parameters and return clause

}

**type class @trait** B2 **extends** A {

**function @abstract** f(); // overrides A.f()

}

**type class** C2 **extends** A, B2 { // legal, as B2 does not clash with A

function f() { ... } // implements A.f() and B2.f()

}

**type class** C3 **extends** A {

function f() { ... } // implements A.f()

}

**type class** D **extends** C2, C3 {

// illegal, as it only one non-trait class can be inherited

}

**type class** E **extends** A, C2 {

// legal, but inheriting A is redundant

}

#### 5.1.1.1 Scope rules

Class constitutes a scope unit. For the uniqueness of identifiers, the rules specified in 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, 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 getters 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.

#### 5.1.1.3 External classes

A class may also be declared as external. In that case, it may declare external member functions without a function body. It is allowed to omit the external keyword from these function declarations. External classes can extend non‑external classes but classes not declared as external shall not extend from external classes. External classes may also define other members like normal classes. When instantiating an external class, the external object being created is provided by the platform adapter and the external method calls to the external 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 in 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.

***Restrictions***

1. Void
2. Void
3. Void
4. An internal class shall not extend an external class

EXAMPLE:

type class @abstract Collection {

function @abstract size() return integer;

// internal default implementation

function isEmpty() return boolean {

return size() == 0

}

}

type external class Stack extends Collection {

function push(integer v);

function pop() return integer;

function isEmpty() return boolean; // external implementation overrides internal

function size() return integer; // external implementation of abstract function}

#### 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* , }\* ")"

**[ 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** and **var template** field or automatic property of the class itself unless they are declared with the **@internal** modifier and also all **const** or **template** fields with no initializer in their order of declaration with the same type as in the declaration. If a **var** or **var template** field has an initializer, the additional formal **in** parameter created for it, for the implicit constructor, shall have the initializer value as the default value of the formal parameter.

NOTE 1: Having a default value in the implicit constructor for the **var** and **var** **template** fields with initializer, makes it possible to skip that parameter when invoking the implicit constructor, or to override it with another value if explicitly provided.

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

}

}

EXAMPLE 3:

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

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

v**ar** **float** b;

// implicit constructor:

// only using variable fields with and without initializer

//**create**(**integer** a := 1, **float** b) {

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

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

//}

}

**type** **class** MySubClassWithDefault **extends** MySuperClass {

**var** **float** f := 1.0;

// implicit constructor:

// **create**(**integer** a := 1, **float** b, **float** f := 1.0) : MySuperClass(a, b) {

// **this**.f := f;

// }

}

#### 5.1.1.6 Constructor invocation

***Syntactic Structure***

*ClassReference* "." **create** [ *ActualParList* ] [ **external** *ActualParList* ]

***Semantic Description***

To instantiate on object, the constructor of the class is invoked. The result of that operation is a reference to a newly constructed of the given concrete class.

If the constructor is a constructor of an external class that has an external formal parameter list, an additional external actual parameter list is given following the external keyword. If the constructor is to be invoked with a parameter list with no actual parameters, then the whole actual parameter list may be omitted.

If the constructor of an external class is invoked, first the external object is created using the given external formal parameters, then the internal constructor is evaluated to initialize the internal part of the object.

EXAMPLE:

**type class** Named {

**var charstring** name;

}

**type external class** Address **extends** Named {

**create**(**charstring** name)

**external** (**charstring** host, **int** portNr)

: Named(name){}

}

**type external class** UnnamedAddress {

**create** (**charstring** host, **int** portNr);

}

**var** Address v\_addr := Address.**create**("Connection 1") **external** ("127.0.0.1", 555);

**var** UnnamedAddress := UnnamedAddress.**create**("127.0.0.1", 555);

**var** Stack v\_stack := Stack.**create**(); // only implicit external constructor without parameters

// calling implicit constructor with default values

**var** MySubClassWithDefault v\_mysub1 := MySubClassWithDefault.**create**(1, 1.0, 1.0);

**var** MySubClassWithDefault v\_mysub2 := MySubClassWithDefault.**create**(1, 1.0);

**var** MySubClassWithDefault v\_mysub3 := MySubClassWithDefault.**create**(b := 1.0);

#### 5.1.1.7 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.

#### 5.1.1.8 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.

#### 5.1.1.9 Method invocation

***Syntactical Structure***

[(ObjectInstance | "super") "."] 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.

The super keyword shall only be used from inside a class member definition to access one of the accessible methods inherited from the super class of the member's containing class.

#### 5.1.1.10 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.

#### 5.1.1.11 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**;

// Indicates whether some object is semantically equivalent to this one,

// according to some equivalence relation.

// The default implementation returns true if and only if both this and obj

// are the same object instance, otherwise returns false.

**public function** equals(object obj) **return boolean** {

**return** this == obj;

}

}

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

EXAMPLE:

**type class** Square {

**public** **function** getSideLength() **return** **integer** { … }  
 …  
 **public function** equals(object obj) **return boolean** {  
 **if** (**this** == obj) {  
 **return** **true**; }  
 **else** **if** (obj **of** Rectangle) {  
 // a rectangle is a square its width and height are of equal lengths  
 **var** Rectangle rectangle := obj => Rectangle;  
 **return** rectangle.getWidth() == getSideLength() **and**   
 rectangle.getHeight() == getSideLength();  
 }  
 **else** **if** (obj **of** Square) {  
 **return** getSideLength() == obj=>Square.getSideLength();  
 }  
 **return false**;  
 }  
}

**type class** Rectangle {  
…  
  
 **public function** getWidth() **return integer** { … }  
 **public function** getHeight() **return integer** { … }  
 **public function** equals(object obj) **return boolean** {  
 **if** (**this** == obj) {  
 **return** **true**; }  
 **else** **if** (obj **of** Square) {  
 **return** obj.**equals**(**this**)**;**  
 }  
 **else** if (obj **of** Rectangle) {  
 **var** Rectangle r2 := obj => Rectangle;  
 **return** getWidth() == r2.getWidth() **and** getHeight() == r2.getHeight();  
 }  
 **return** **false**;  
 }  
}

#### 5.1.1.12 Properties

***Syntactic Structure***

**var** [ *TemplateModifier* ] { **@abstract** | **@final** | **@deterministic** | **@internal**}

*Type* @**property** *Identifier*

[ ":=" *TemplateBody* ]

[ **"{"**

{

[**public** | **private**]{ **@abstract** | **@final** | **@deterministic** }

**@get** [ "=>" *TemplateBody* | *StatementBlock* ] [";"]

|

[**public** | **private**] { **@abstract** | **@final** | **@deterministic }**

**@set** [ "=>" *Assignment* | *StatementBlock* ] [";"]

}

"}"

]

***Semantic Description***

A class property is a class var 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). It can be syntactically mixed in a compound var declaration with other variables.

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

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.

Visibility can be declared on the property declaration level or before the getter and setter declarations inside the property body separately. If no visibility is declared before a getter or setter directly, the visibility of the property declaration is used for it.

While most properties are declared with a property body containing either a getter or setter function, it is allowed to declare a property without a body. Such a property is called an automatic property. Automatic properties are similar to member variables, they are always generated with an associated 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 with the name of the automatic property as the name of the formal parameter.

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 clause 5.4.2 of [1]. The **value** variable of the setter function has the same template modifiers as the property itself.

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.

***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. An 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 clause 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 clause 15.8 of [1].
3. When passing a value to the value variable of the setter function, the rules specified in clause 5.4.2 of [1] shall apply.
4. If a property or its getter or setter contain the **@deterministic** modifier, rules for derministic functions specified in clause 16.1.0 of [1] shall apply to the body of the concerned getter or setter.
5. An error shall be produced when a property that has no getter is referenced on the right hand side of an assignment.
6. An error shall be produced when a property that has no setter is referenced on the left hand side of an assignment.
7. 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 executing a **return** statement or a statement that terminates component execution (such as **stop** or **testcase.stop**).
8. The 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 resolve into a template that fulfills the restrictions set by the *TemplateModifier* of the property. These restrictions are specified in clause 15.8 of [1].
9. 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.
10. An error shall be produced if an index or dot notation is applied to a property referenced on the left hand side of an assignment.

EXAMPLE:

**type class** Rectangle {

**private var integer** heightVal;

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

**public @property integer** 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 integer** perimeter {

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

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

}

}

}

…

**var** Rectangle v\_rect := Rectangle.**create**(heightVal := 10, width := 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

v\_rect.height := -100; // does not change the heightVal variable

### 5.1.2 Objects

#### 5.1.2.0 General

Objects are the instances of classes. Each instance comprises an instance of the data of the fields of the class (including all superclasses) and allows invocation of its public methods by other behaviour and protected or private methods by behaviour defined by the object's class itself.

#### 5.1.2.1 Ownership

Each object is owned by the component on which it was created. The owning component of an object can be referenced via the self component reference. Methods of objects can only be invoked by behaviour that also runs on the owning component. An object is created on a component if its constructor was invoked by a behaviour running on that component.

#### 5.1.2.2 Object References

Objects are always passed by reference (even though their formal parameters can still be in, inout or out, dependent on the usage of that parameter). A variable of a class type contains only a reference to the object instance and the object is not copied when used as an actual parameter or assigned to a variable, but only the reference to the object. Therefore, multiple variables can contain a reference to the same object simultaneously.

***Restrictions***

1. Object References shall not be passed as actual parameter or part of an actual parameter to either the create operation of a component type or a function started on a component.

NOTE: Since objects cannot be shared by different component contexts and for each component at most one behaviour is running, no parallel conflicting access to any of the objects fields or methods is possible.

EXAMPLE:

**type class** MyClass() {

**var** **integer** a := 0;

**function** increment() {

a := a + 1;

}

**function** getter() **return integer** {

**return** a;

}

}

**type record of** MyClass ROC;

**…**

**var** MyClass v\_a := MyClass.**create**();

**var** ROC my\_roc := {v\_a};

**var** ROC my\_roc2 := my\_roc; //create a copy

v\_a.increment();

my\_roc[0].increment();

my\_roc2[0].increment();

**var integer** v\_temp1 := my\_roc[0].getter(); // returns 3

**var integer** v\_temp2 := my\_roc2[0].getter(); // returns 3

#### 5.1.2.3 Null reference

An object variable that is not initialized with an object instance contains the special value null. An object variable or parameter may be compared with the special value null with the equality and inequality operators or can be assigned the special value null explicitly.

#### 5.1.2.4 Select class-statement

***Syntactical Structure***

**select** **class** "(" *Object* ")"   
"{" { **case** "(" *ClassReference* ")" *StatementBlock* }+ [*ElseCase*] "}"

***Semantic Description***

The class of an object can be discriminated for via the 'select class' statement that is similar to a select union statement insofar that it allows only superclasses and known subclasses of the object reference's class in the context. If more than one case contains a superclass of the actual class of the given object instance, the first of these cases will be chosen by the select class statement.

In case that the *Object* is not an instance of any of the *ClassReference*s in the different cases, the statement block in the *ElseCase*, if present, will be executed.

EXAMPLE:

**type class** A {}

**type class** B **extends** A {}

…

**var** A v\_a := B.**create**();

**select class** (v\_a) {

**case** (B) { … } // will be chosen

**case** (A) { … } // will not be chosen

}

***Restrictions***

1. If a class from one case is a superclass of a class from another case, then the case of the subclass shall be precede the case of the superclass.

#### 5.1.2.5 Of-operator (Dynamic Class Discrimination)

***Syntactical Structure***

Object **of** ClassReference

***Semantic Description***

To check whether an object is an instance is of a certain class, the of operator may be used.

It yields a Boolean value which is true if and only if the most specific class of the object referenced on the left-hand side is either equal to or a subclass derived from the class type reference on the right-hand side.

#### 5.1.2.6 Casting

***Syntactical Structure***

ObjectReference "=>" ( ClassIdentifier | " (" ClassReference ")" )

***Semantic Description***

An object reference can be cast to another class of the object's known class's set of direct or indirect superclasses and direct or indirect subclasses. This operation yields an object reference to the same object but can be used as being of the type being cast to. If the referenced class to be cast to is an expression that is not a simple identifier, the expression shall be written in parenthesis.

***Restrictions***

1. If the class the object is being cast to is not in the set of superclasses or the concrete class of the object, the cast operation shall result in an error.

#### 5.1.2.7 Comparison

***Syntactical Structure***

*ObjectReference* "==" *ObjectReference* |

*ObjectReference* "!=" *ObjectReference*

***Semantic Description***

Two object references can be compared for equality or inequality. Two object references are equal if and only if they reference the same object instance. An object reference is equal to the special null object reference if it does not reference an object instance.

***Restrictions***

1. Object references can only be compared if they are type compatible, i.e. if they both have the same declared class type or if the declared type of one of the references is a direct or indirect superclass of the declared type of the other reference.

***Examples***

**type class** Shape {}

**type class** Circle **extends** Shape {}

**type class** Rectangle **extends** Shape {}

**var** Circle v\_circle := Circle.**create**();

**var** Rectangle v\_rectangle := Rectangle.**create**();

**var** Shape v\_shape := v\_circle;

v\_shape == v\_circle // allowed => true

v\_circle != v\_rectangle // not allowed

v\_shape == **null** // allowed => false

v\_shape == Circle.**create**() // allowed => false

#### 5.1.2.8 Object Templates

***Syntactical Structure***

"{" { ( *FieldName* | *FunctionInstance* ) ":=" *TemplateBody* [","] } "}"

***Semantic Description***

An object template is a matching mechanism to be used for objects similar to those used for record values. It can be used as a *TemplateBody* in template declarations for templates of class type and everywhere else where a *TemplateBody* is acceptable except in sending and receiving operations as objects can neither be sent nor received.

If an object template is used in a matching operation to match against an object reference, it matches if and only if for all assignments in the compound assignment notation the matching mechanism on the right hand side of the assignment matches the value yielded by the evaluation of applying the left hand side of the assignment as dotted notation to the object.

NOTE 1: It is not necessary to add an assignment in the template for *all* public properties that exist in the class. For all public properties that are not assigned any matching mechanism in the template, a "don't care" semantics is assumed. This is especially useful for using templates for objects of subclasses that might have additional properties.

Object templates declared for a class can also be used to match against objects of all subclasses.

The modifies operation is allowed also for Object templates with the same procedure as for record templates, but only *FieldName* assignments are modified while *FunctionInstance* assignments are not modified, though additional FunctionInstance assignments may be added.

It is allowed to have multiple *FunctionInstance* assignments of the same method in the same object template, possibly with different actual parameters.

NOTE 2: Usage of *FunctionInstance* assignments can lead to contradictory assignments that would lead to the object template not matching any object of the class.

***Restrictions***

1. The *FieldName* in an assignment in the template shall be the name of a public value property of the template's class. The type of the property shall be compatible with the corresponding *TemplateBody* on the right hand side.
2. The name of the function in the *FunctionInstance* in an assignment in the template shall be the name of a public method of the template's class. The actual parameter lists given in the *FunctionInstance* shall be compatible with the formal parameter lists of that method and the return type of the function shall be a value type compatible with the corresponding *TemplateBody* on the right hand side.
3. The getter of a public property or the function used on the left hand side of an assignment in an object template shall be deterministic and shall fulfill the restrictions imposed on content of functions used in special places given in clause 16.1.4.
4. The names in *FieldName* assignments in an object template shall be unique, i.e. no *FieldName* shall appear more than once on the left hand side.
5. Object templates shall not be used in sending or receiving operations.
6. Object template shall not be used as values and can not be converted to a value with the **valueof** operation.

***Examples***

EXAMPLE 1:

**type class** Pair { **public var integer @property** a, **@property** b }

**template** Pair t := { a := (1 .. 20) }

**type class** Triple **extends** Pair { **public var integer @property** c }

**match**(Triple.**create**(1,2,3), t) // returns true

EXAMPLE 2:

**type** **class** @**abstract** Shape { **public** **function** @**abstract** area() **return** **float**; }

// smallShape would match for all objects whose class is derived from Shape

// and where the result of the method call to area() fulfills the constraint.

**template** Shape smallShape := { area() := (0.0 .. 20.0) }

// contradictory template:

**template** Shape empty := {

area() := (0.0 .. **infinity**),

area() := {-**infinity** .. !0.0)

}

### 5.1.3 Extension to ETSI ES 201 873-1, clause 7.1.8 (Presence checking operators)

**Clause 7.1.8.0 General**

The presence checking operators (**ispresent**, **ischosen**, **isvalue** and **isbound)** shall also apply to object references and invocations of methods of objects.

If resolving a dot notation on an object would produce an error (the object referenced or returned by the invoked method having the null value) the following happens:

* No error is produced.
* Evaluation of all remaining unresolved fields in the *ExtendedFieldReference* is stopped. All remaining parts of the *ExtendedFieldReference* that are located right from the operation that would normally produce an error up to the end of the presence checking operator are not evaluated.
* The presence checking operator yields the value **false**.

The rule on special handling of dot notation, index notation, object method invocation and decoded field references is not applied recursively. Errors occurring during the invocation of an object's method are not affected by this rule.

**Clause 7.1.8.1 The ispresent operator**

The **ispresent** operator can also be used to check if an object instance is present.

For an object instance argument the **ispresent** operator returns:

* The value **false** if the object is set to the null value.
* The value **true** otherwise.

***Examples***

EXAMPLE:

// Given

**type class** A {}

**var** A v\_a1;

**var** A v\_a2 := A.**create**();

**var** **boolean** v\_checkResult := **ispresent**(v\_a1) // yields false  
v\_checkResult := **ispresent**(v\_a2) // yields true

**Clause 7.1.8.3 The isvalue operator**

The **isvalue** operator always returns the value true for an object instance.

***Examples***

EXAMPLE:

// Given

**type class** A {}

**var** A v\_a1;

**var** A v\_a2 := A.**create**();

**var** **boolean** v\_checkResult := **isvalue**(v\_a1) // yields true  
v\_checkResult := **isvalue**(v\_a2) // yields true

**Clause 7.1.8.4 The isbound operator**

The **isbound** operator always returns the value true for an object instance.

***Examples***

EXAMPLE:

// Given

**type class** A {}

**var** A v\_a1;

**var** A v\_a2 := A.**create**();

**var** **boolean** v\_checkResult := **isbound**(v\_a1) // yields true  
v\_checkResult := **isbound**(v\_a2) // yields true

## 5.2 Exception handling

### 5.2.0 General

This clause introduces exception handling into TTCN-3. It provides means to define exception handling for functions, external functions, altsteps and test cases.

### 5.2.1 Extension to ETSI ES 201 873-1, clause 16.1.0 (Functions)

**Clause 16.1.0 General**

The syntax of functions is extended with an optional **exception** clause.

***Syntactical Structure***

**function** [ **@deterministic** | **@control** ] *FunctionIdentifier*

"(" { ( *FormalValuePar* | *FormalTemplatePar* ) [","] } ")"

[ **runs on** *ComponentType* ]

[ **mtc** *ComponentType* ]

[ **system** *ComponentType* ]

[ **return** [ template ] *Type* ]

[ **exception** "(" {Type [","]}+ ")" ]

*StatementBlock*

**Clause 16.1.0 General**

The semantic description part is extended.

Functions may have an exception list. The exception list declares, what exception types may be raised during the execution of the function either directly or indirectly.

NOTE 1: The exception list can be used to communicate to the callers of the function what exceptions to prepare for and by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 2: The exception list might not be exhaustive. With activated altsteps it might not be possible to precisely know what exceptions might be raised within a function directly or indirectly.

If the statement block of a function has a **finally** block, the finally block is always executed before control returns to the location of the call of the function.

### 5.2.2 Extension to ETSI ES 201 873-1, clause 16.1.3 (External Functions)

**Clause 16.1.3 General**

The syntax of external functions is extended with the optional **exception** clause.

***Syntactical Structure***

**external function** [ **@deterministic** | **@control** ] *ExtFunctionIdentifier*

"(" { ( *FormalValuePar* | *FormalTemplatePar* ) [","] } ")"

[ **return** [ **template** [ *Restriction* ] ] *Type* ] [ **exception** "(" {Type [","]}+ ")" ]

**Clause 16.1.3 General**

The semantic description part is extended.

External functions may have an exception list. The exception list declares, what exception types may be raised during the execution of the external function.

NOTE 0: The exception list can be used by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 1: The exception list might not be exhaustive. It might not be possible to precisely know what exceptions might be raised within an external function directly or indirectly.

### 5.2.3 Extension to ETSI ES 201 873-1, clause 16.1.4 (Invoking functions from specific places)

**Clause 16.1.4 General**

The list of restrictions is extended to avoid side effects.

n) Raising an exception with the raise exception statement.

### 5.2.4 Extension to ETSI ES 201 873-1, clause 16.2 (Altsteps)

**Clause 16.2.0 General**

The syntax of altstep is extended with the optional **exception**, **catch** and **finally** clauses.

***Syntactical Structure***

**altstep** [ **@control** ] [ **interleave** ] AltstepIdentifier

"(" { ( FormalValuePar | FormalTemplatePar) [","] } ")"

[ **runs** **on** ComponentType ]

[ **mtc** ComponentType ]

[ **system** ComponentType ]

[ **exception** "(" {Type [","]}+ ")" ]

"{"

{ ( VarInstance | TimerInstance | ConstDef | TemplateDef ) [";"] }

AltGuardList

"}"

{ CatchBlock }

[ FinallyBlock ]

**Clause 16.2.0 General**

The semantic description part is extended.

Altsteps may have an exception list. The exception list declares, what exception types may be raised during the execution of the altstep either directly or indirectly.

NOTE 0: The exception list can be used to communicate to the callers of the altstep what exceptions to prepare for and by tools to perform stronger static checks. For backward compatibility reasons the exception list is optional.

NOTE 1: The exception list might not be exhaustive. With activated altsteps it might not be possible to precisely know what exceptions might be raised within an altstep directly or indirectly.

Altsteps may have a finally block. If present the finally block is always executed before control returns to the location of the call of the altstep every time the altstep is invoked, regardless of whether implicitly as an activated default or explicitly from another alt statement.

### 5.2.5 Extension to ETSI ES 201 873-1, clause 16.3 (Test cases)

**Clause 16.3 Test cases**

The semantic description part is extended.

The StatementBlock of Test cases may have a finally block. If present the finally block is always executed before the test case terminates.

Exceptions raised directly or indirectly within the test case and not handled latest by the catch clauses of the StatementBlock of the testcase results in the testcase finishing with a dynamic error. In this situation the dynamic error has to reference not handling the exception as the reason of error.

NOTE 0: The reason for the dynamic error is not the raising of the exception, but the lack of handling within the testcase.

### 5.2.6 Extension to ETSI ES 201 873-1, clause 18 (Overview of program statements and operations)

The list of statements in table 15 of ETSI ES 201 873-1 needs to be extended with a raise exception statement as shown below.

| Statement | Associated keyword or symbol | Can be directly or indirectly invoked by module control, but not by test components | Can be invoked by functions, test cases and altsteps running on test components | Can be directly or indirectly invoked from specific places (see note 1) |
| --- | --- | --- | --- | --- |
| Expressions | **(…)** | Yes | Yes | Yes |
| **Basic program statements** | | | | |
| Assignments | **:=** | Yes | Yes | Yes (see note 4) |
| If-else | **if (…) {…} else {…}** | Yes | Yes | Yes |
| Select case | **select case (…) { case (…) {…} case else {…}}** | Yes | Yes | Yes |
| Select class | **select class (…) { case (…) {…} case else {…}}** | Yes | Yes | Yes |
| For loop | **for (…) {…}** | Yes | Yes | Yes |
| While loop | **while (…) {…}** | Yes | Yes | Yes |
| Do while loop | **do {…} while (…)** | Yes | Yes | Yes |
| Label and Goto | **label / goto** | Yes | Yes | Yes |
| Stop execution | **stop** | Yes | Yes |  |
| Returning control | **return** |  | Yes (see note 5) | Yes |
| Leaving a loop, alt, altstep or interleave | **break** | Yes | Yes | Yes |
| Next iteration of a loop | **continue** | Yes | Yes | Yes |
| Raise exception | **raise** | Yes | Yes | Yes |
| Logging | **log** | Yes | Yes | Yes |
| **Statements and operations for alternative behaviours** | | | | |
| Alternative behaviour | **alt {…}** | Yes (see note 2) | Yes |  |
| Re-evaluation of alternative behaviour | **repeat** | Yes | Yes |  |
| Interleaved behaviour | **interleave {…}** | Yes (see note 2) | Yes |  |
| Activate a default | **activate** | Yes | Yes |  |
| Deactivate a default | **deactivate** | Yes | Yes |  |
| **Configuration operations** | | | | |
| Create parallel test component | **create** |  | Yes |  |
| Connect component port to component port | **connect** |  | Yes |  |
| Disconnect two component ports | **disconnect** |  | Yes |  |
| Map port to test interface | **map** |  | Yes |  |
| Unmap port from test system interface | **unmap** |  | Yes |  |
| Get MTC component reference value | **mtc** |  | Yes | Yes |
| Get test system interface component reference value | **system** |  | Yes | Yes |
| Get own component reference value | **self** |  | Yes | Yes |
| Start execution of test component behaviour | **start** |  | Yes |  |
| Stop execution of test component behaviour | **stop** |  | Yes |  |
| Terminating the testcase with an error verdict | **testcase.stop** |  | Yes | Yes |
| Remove a test component from the system | **kill** |  | Yes |  |
| Check termination of a PTC behaviour | **running** |  | Yes |  |
| Check if a PTC exists in the test system | **alive** |  | Yes |  |
| Wait for termination of a PTC behaviour | **done** |  | Yes |  |
| Wait a PTC cease to exist | **killed** |  | Yes |  |
| **Communication operations** | | | | |
| Send message | **send** |  | Yes |  |
| Invoke procedure call | **call** |  | Yes |  |
| Reply to procedure call from remote entity | **reply** |  | Yes |  |
| Raise exception (to an accepted call) | **raise** |  | Yes |  |
| Receive message | **receive** |  | Yes |  |
| Trigger on message | **trigger** |  | Yes |  |
| Accept procedure call from remote entity | **getcall** |  | Yes |  |
| Handle response from a previous call | **getreply** |  | Yes |  |
| Catch exception (from called entity) | **catch** |  | Yes |  |
| Check (current) message/call received | **check** |  | Yes |  |
| Clear port queue | **clear** |  | Yes |  |
| Clear queue and enable sending & receiving at a to port | **start** |  | Yes |  |
| Disable sending and disallow receiving operations to match at a port | **stop** |  | Yes |  |
| Disable sending and disallow receiving operations to match new messages/calls | **halt** |  | Yes |  |
| Check the state of a port | **checkstate** |  | Yes |  |
| **Timer operations** | | | | |
| Start timer | **start** | Yes | Yes |  |
| Stop timer | **stop** | Yes | Yes |  |
| Read elapsed time | **read** | Yes | Yes |  |
| Check if timer running | **running** | Yes | Yes |  |
| Timeout event | **timeout** | Yes | Yes |  |
| **Verdict operations** | | | | |
| Set local verdict | **setverdict** |  | Yes |  |
| Get local verdict | **getverdict** |  | Yes | Yes |
| **External actions** | | | | |
| Stimulate an (SUT) action externally | **action** | Yes | Yes |  |
| **Execution of test cases** | | | | |
| Execute test case | **execute** | Yes | Yes (see note 3) |  |
| NOTE 1: Specific places are defined in clause 16.1.4. Only operations that do not have any potential side effects on snapshot evaluation are allowed.  NOTE 2: Can be used to control timer operations only.  NOTE 3: Can only be used in functions and altsteps that are used in module control.  NOTE 4: Changing of component variables is disallowed.  NOTE 5: Can be used in functions and altsteps but not in test cases. | | | | |

### 5.2.7 Extension to ETSI ES 201 873-1, clause 19 (Basic program statements)

**Clause 19.0 General**

The list of statements in table 17 needs to be extended with the raise exception statement as described below.

|  |  |
| --- | --- |
| Basic program statements | |
| Statement | Associated keyword or symbol |
| Assignments | **:=** |
| If-else | **if (…) {…} else {…}** |
| Select case | **select case (…) { case (…) {…} case else {…}}** |
| Select class | **select class (…) { case (…) {…} case else {…}}** |
| For loop | **for (…) {…}** |
| While loop | **while (…) {…}** |
| Do while loop | **do {…} while (…)** |
| Label and Goto | **label / goto** |
| Stop execution | **stop** |
| Returning control | **return** |
| Leaving a loop, alt, altstep or interleave | **break** |
| Next iteration of a loop | **continue** |
| Raise exception | **raise** |
| Logging | **log** |

**Clause 19.14 Statement Block**

The syntax of statement block is changed as shown below.

BasicStatementBlock: "{" { *LocalDefinition* | *Statement* } "}"

StatementBlock: BasicStatementBlock {**catch** "(" Type Identifier ")" Basic*StatementBlock* }

[**finally** Basic*StatementBlock*]

**Clause 19.14 Statement Block**

The semantic description part is extended.

A basic statement block is a sequence of declarations and statements.

Statement blocks can be used like basic program statements to introduce a local scope in the flow of control of TTCN-3 behaviour. The declarations and statements in a basic statement block are executed in the order of their appearance, i.e. sequentially.

A statement block consists of a basic statement block with optional additional catch clauses, that can be used to handle exceptions raised directly or indirectly within the basic statement block and an optional finally clause which is executed after the basic statement block execution. When an exception is raised by a statement in the basic statement block the catch clauses are tried in order of appearance to find one of the same type for data types or one the exception can be cast to if it is a type class kind exception. Execution continues with the basic statement block of the first catch clause whose type matches the type of the raised exception.

The catch clause declares a variable of an exception, with the type and identifier provided, to hold the value of the exception within the catch clause. The scope of this variable is limited to the basic statement block of the catch clause, i.e. it is only visible inside the body of the catch clause.

NOTE: The scope of the catch and finally blocks is on the same level with the scope of the basic statement block. Local variables declared within the basic statement block are not visible in the catch and finally clauses.

**Clause 19.14 Statement Block**

The list of restrictions is extended:

1. The control transfer statements **return**, and **raise** shall not be used in the **finally** clause. Functions that can raise exceptions shall not be called in the finally clause.
2. The basic statement block of a **catch** clause shall obey the same semantic restrictions as the basic statement block it follows.

Clause 19 is extended with a new clause.

**NEW: Clause 19.15 The Raise exception statement**

The **raise** exception statement raises an exception, causing the execution to continue at the catch block closest in the procedure call hierarchy, also executing all **finally** blocks it encounters while traversing the procedure call hierarchy.

***Syntactical Structure***

**raise** TemplateInstance

***Semantic Description***

The **raise** statement is used to raise an exception. On executing a **raise** exception statement the statement blocks, loops, **alt** statements or **interleave** statement within the encompassing function/altstep/testcase are left. If the encompassing function, altstep or testcase has a catch block with the exact same type as that of the raised exception value for data types or one the exception can be cast to if it is a class type exception, execution continues in that catch block. If the encompassing function or altstep does not have catch blocks or none of the catch blocks can handle the raised exception, execution leaves the function or altstep to handle the exception in the calling function, altstep or testcase. An exception not handled via catch clause of the StatementBlock of a testcase shall cause a dynamic error.

If the StatementBlock of a function, altstep or testcase has a finally block, this finally block is always executed before the function, altstep, testcase terminates. If an exception was raised and handled in a catch block, the finally block is executed after the catch block. If there was no exception raised, or an exception was raised but not handled in any catch blocks the finally block is executed before the function, altstep or testcase terminates.

The parameter of the **raise** operation shall evaluate to a value, that the exception will have.

Exceptions are specified as types. Therefore the exception value may either be derived from a template conforming to the template(value) restriction or be the value resulting from an expression (which of course can be an explicit value). The type of the value specification to the **raise** operation shall be determinable as it is necessary to avoid any ambiguity of the type of the value being raised.

NOTE 0: The type of the raised exception should be provided explicitly for literal values. Catch clauses with synonym types or restricted types will only catch exceptions of the same type.

***Restrictions***

In addition to the general static rules of TTCN‑3 given in clause 5 and shown in table 15, the following restrictions apply:

1. An exception shall only be raised inside a function, altstep or testcase.
2. The *TemplateInstance* shall conform to the template(value) restriction (see clause 15.8).
3. Exceptions shall not be raised directly or indirectly inside finally blocks of functions, altsteps or testcases.

***Examples***

EXAMPLE 1:

**raise** ( v\_myVariable + v\_yourVariable - 2);

// Raises an exception with a value which is the result of the arithmetic expression

**raise** **integer**:5; // Raises an exception with the integer value 5

**raise** **charstring**:"Olala!";

// Raises an exception with the charstring value "Olala!"

EXAMPLE 2: Catching an exception raised in a called function.

**type** **record** **of** **charstring** t\_registeredNames;

**type** **component** myComponent {

**var** t\_registerdNames v\_registeredNames;

}

**function** f\_init(**in** **charstring** name) **exception** (**charstring, integer**) **runs on** myComponent

{

...

**if** (name\_was\_not\_registered) {

**raise** ("Could not initialize " & name); // when the exception is raised f\_init terminates

}

...

}

**function** f\_operation(**in** **charstring** user1, **in** **charstring** user2) **exception** (**integer**)

**runs on** myComponent {

f\_init(user1);

f\_init(user2);

...

} **catch** (**charstring** e) {

// the exception is available for processing in the e variable

// release resources and terminate function

} **catch** (**integer** e) {

//there was some other issue

// release resources

**raise** e; /// the exception is raised again to be handled in the calling function

}

EXAMPLE 3: Finally is always executed.

**function** f\_operation2(**in** **charstring** user1, **in** **charstring** user2) **exception** (**charstring**)

**runs on** myComponent {

f\_init(user1);

f\_init(user2);

…

} **finally** {

// finally is executed whether there was an exception or not before the function terminates

}

EXAMPLE 4: The exception can travel through several functions in the call hierarchy until handled.

**function** f\_operation3(**in** **charstring** user1, **in** **charstring** user2) **exception** (**charstring**)

**runs on** myComponent {

f\_operation2(user1, user2); // an exception is raised in f\_init

…

} **finally** {

// after the finally block in f\_operation2 this finally block is also executed

// the exception is not caught.

}

EXAMPLE 5: Exception not caught latest in a testcase is reported as dynamic error.

**testcase** t\_myTest1() **runs on** myComponent {

f\_init("user1");

f\_init("unknown user");// bad argument will raise an exception in f\_init

… // because of the raised exception execution continues in the finally block

} **finally** {

… // via the runs on component resources can be freed

// as the exception is not caught dynamic error is reported

}

EXAMPLE 6: The type of the exception has to match the type of the catch clause exactly.

**function** f\_example() **exception** (**integer**) {

**raise** **integer**:5;

}

**type** **integer** MyIntegerSynonim;

**type** **integer** MyIntegerRange (0 .. 255);

**function** f\_example2(){

f\_example();

} **catch** (MyIntegerRange e) {

// The exception is not caught here.

// The type of the raised exception and the type of the catch type has to be the same

} **catch** (MyIntegerSynonim e) {

// The exception is not caught here.

// The type of the raised exception and the type of the catch type has to be the same

} **catch** (**integer** e) {

// As the exception raised in f\_example was raise with the integer type it is handled here

}

# 6 TRI Extensions for the Package

## 6.1 Extensions to clause 5.3 of ETSI ES 201 873-5 Data interface

**Clause 5.3.2 Communication**

**The clause is to be modified:**

TriExceptionType A value of type TriExceptionType is an encoded type and value of an exception that either is to be sent to the SUT or has been received from the SUT. This abstract type is used in procedure based TRI communication operations and raising exception during execution of external functions, constructors, destructors and methods.

**Clause 5.3.4 Miscellaneous**

**The clause is to be extended:**

TriClassIdType A value of type TriClassIdType is the name of a class as specified in the TTCN‑3 ATS.

TriObjHandleType A value of type TriObjHandle contains platform-specific data allowing access to external objects.

## 6.2 Extensions to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations

**Clause 5.6.3.4 triExternalCreate (TE** → **PA)**

**This clause is to be added.**

|  |  |
| --- | --- |
| **Signature** | TriStatusType triExternalCreate(  in TriClassIdType classId,  inout TriParameterListType parameterList, out TriObjHandleType createdObject) |
| **In Parameters** | classId identifier of the external class |
| **Out Parameters** | returnValue handle to the created object |
| **InOutParameters** | parameterList a list of encoded parameters for the indicated constructor. The parameters in parameterList are ordered as they appear in the TTCN‑3 constructor declaration. |
| **Return Value** | The return status of the triExternalCreate operation. The return status indicates the local success (***TRI\_OK***) or failure (***TRI\_Error***) of the operation. |
| **Constraints** | This operation is called by the TE when it invokes a constructor specified in a class which is defined as external in TTCN‑3.  In the invocation of a triExternalCreate operation by the TE all *in* and *inout* constructor parameters contain encoded values. No error shall be indicated by the PA in case the value of any *out* parameter is non‑null. |
| **Effect** | The PA shall implement the behaviour for each external class specified in the TTCN‑3 ATS. On invocation of this operation, the PA shall invoke the constructor of a class indicated by the identifier classId. It shall access the specified *in* and *inout* constructor parameters in parameterList, create a new external object instance using the values of these parameters, and compute values for *inout* and *out* parameters in parameterList. The operation shall then return encoded values for all *inout* and *out* constructor parameters and a handle to the created external object.  The triExternalCreate operation returns ***TRI\_OK*** if the PA completes the constructor of the external class successfully, ***TRI\_Error*** otherwise. In the latter case, the distinct value null shall be returned as the object handle.  Note that whereas most of other TRI operations are considered to be non‑blocking, the triExternalCreate operation is considered to be *blocking.* That means that the operation shall not return before the construction of the external object has been finished. External constructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. |

**Clause 5.6.3.5 triExternalFinally (TE** → **PA)**

**This clause is to be added.**

|  |  |
| --- | --- |
| **Signature** | TriStatusType triExternalFinally(  in TriObjHandleType handle) |
| **In Parameters** | handle handle to the object being destroyed |
| **Return Value** | The return status of the triExternalFinally operation. The return status indicates the local success (***TRI\_OK***) or failure (***TRI\_Error***) of the operation. |
| **Constraints** | This operation is called by the TE when it invokes a destructor specified in a class which is defined as external in TTCN‑3. |
| **Effect** | The PA shall implement the behaviour for each external class specified in the TTCN‑3 ATS which contains a destructor definition. On invocation of this operation, the PA shall invoke the destructor of the object whose handle is in the handle parameter.  The triExternalFinally operation returns ***TRI\_OK*** if the PA completes destruction of the external object successfully, ***TRI\_Error*** otherwise.  Note that whereas most of other TRI operations are considered to be non‑blocking, the triExternalFinally operation is considered to be *blocking.* That means that the operation shall not return before the destruction of the external object has been finished. External destructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. |

**Clause 5.6.3.6 triExternalMethod (TE** → **PA)**

**This clause is to be added.**

|  |  |
| --- | --- |
| **Signature** | TriStatusType triExternalMethod(  in TriObjHandleType handle,  in String methodName,  inout TriParameterListType parameterList, out TriParameterType returnValue) |
| **In Parameters** | handle handle of the affected object; null for static methods  methodName name of the called method |
| **Out Parameters** | returnValue (optional) encoded return value |
| **InOutParameters** | parameterList a list of encoded parameters for the indicated method. The parameters in parameterList are ordered as they appear in the TTCN‑3 method declaration. |
| **Return Value** | The return status of the triExternalMethod operation. The return status indicates the local success (***TRI\_OK***) or failure (***TRI\_Error***) of the operation. |
| **Constraints** | This operation is called by the TE when it invokes a method specified in a class which is defined as external in TTCN‑3.  In the invocation of a triExternalMethod operation by the TE all *in* and *inout* parameters contain encoded values. No error shall be indicated by the PA in case the value of any *out* parameter is non‑null. |
| **Effect** | The PA shall implement the behaviour for each method of all external classes specified in the TTCN‑3 ATS. On invocation of this operation, the PA shall call a method methodName of an external object whose handle is in the handle parameter. It shall access the specified *in* and *inout* method parameters in parameterList, pass the values of these parameters to the called method, and compute values for *inout* and *out* parameters in parameterList. The operation shall then return encoded values for all *inout* and *out* method parameters and the encoded return value.  If no return type has been defined for this method in the TTCN‑3 ATS, the distinct value null shall be used for the latter.  The triExternalMethod operation returns ***TRI\_OK*** if the PA completes the method of the external object successfully, ***TRI\_Error*** otherwise.  Note that whereas most of other TRI operations are considered to be non‑blocking, the triExternalMethod operation is considered to be *blocking.* That means that the operation shall not return before the method call has been finished. Methods of external classes have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. |

**Clause 5.6.3.7 triRaiseException (PA** → **TE)**

**This clause is to be added.**

|  |  |
| --- | --- |
| **Signature** | void triExternalRaise(in TriExceptionType exc) |
| **In Parameters** | exc encoded exception to raise |
| **Constraints** | This operation can be called by the PA only during execution of triExternalFunction, triExternalCreate, triExternalFinally or triExternalMethod. |
| **Effect** | The operation raises an exception that can be later processed by the TE in the catch statement. The exception is provided in an encoded form. The TE performs decoding when the triExternalFunction, triExternalCreate, triExternalFinally or triExternalMethod where the exception was raised returns. Decoding is performed in the catch statement. |

## 6.3 Extensions to clause 6 of ETSI ES 201 873-5 JavaTM language mapping

**Clause 6.3.3.20 TriObjHandleType**

This clause is to be added.

**TriClassIdType** is mapped to the following interface:

// TRI IDL TriClassIdType

package org.etsi.ttcn.tri;

public interface TriClassId {

public String toString();

public String getClassName();

public boolean equals(TriClassId id);

}

**Methods:**

* toString  
  Returns the string representation of the class as defined in TTCN‑3 specification.
* getClassName  
  Returns the class identifier as defined in the TTCN‑3 specification.
* equals  
  Compares id with this TriClassId for equality. Returns true if and only if both classes have the same class identifier, false otherwise.

**Clause 6.3.3.20 TriObjHandleType**

This clause is to be added.

**TriObjHandleType** is mapped to the java.lang.Object class.

**Clause 6.3.3.21 TriObjHandleWrapper**

This clause is to be added.

**TriObjHandleWrapper** is used in the triExternalCreate operation as a placeholder for the created object handle.

public interface TriObjHandleWrapper {

public void setHandle(Object handle);

public Object getHandle();

}

**Methods:**

* setHandle  
  Sets the contained object.
* getHandle  
  Gets the contained object.

**Clause 6.5.3.1 TriPlatformPA**

This clause is to be extended.

// TriPlatform

// TE ‑> PA

package org.etsi.ttcn.tri;

public interface TriPlatformPA {

…

// Ref: TRI‑Definition 5.6.3.4

public TriStatus triExternalCreate(TriClassIdType classId,

TriParameterList parameterList, TriObjHandleWrapper handle);

// Ref: TRI‑Definition 5.6.3.5

public TriStatus triExternalFinally(Object handle);

// Ref: TRI‑Definition 5.6.3.6

public TriStatus triExternalMethod(Object handle, String methodName,

TriParameterList parameterList, TriParameter returnValue);

}

**Clause 6.5.3.2 TriPlatformTE**

This clause is to be extended.

// TriPlatform

// PA ‑> TE

package org.etsi.ttcn.tri;

public interface TriPlatformTE {

…

// Ref: TRI-Definition 5.6.3.7

public void triRaiseException(TriException exc);

}

## 6.4 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping

**Clause 7.2.1 Abstract type mapping**

This clause is to be extended.

| TRI ADT | ANSI C Representation | Notes and comments |
| --- | --- | --- |
| … |  |  |
| TriClassIdType | QualifiedName |  |
| TriObjectHandleType | typedef void \* TriObjectHandle; |  |

**Clause 7.2.4 TRI operation mapping**

This clause is to be extended.

| IDL Representation | ANSI C Representation |
| --- | --- |
| **…** |  |
| TriStatusType triExternalCreate (in TriClassIdType classId, inout TriParameterListType parameterList, out TriObjHandleType createdObject) | TriStatus triExternalCreate  (const TriClassId\* classId,  TriParameterList\* parameterList,  TriObjectHandle\* handle) |
| TriStatusType triExternalFinally (in TriObjHandleType handle) | TriStatus triExternalFinally  (TriObjectHandle handle) |
| TriStatusType triExternalMethod(in TriObjHandleType handle, in String methodName, inout TriParameterListType parameterList, out TriParameterType returnValue) | TriStatus triExternalFunction  (TriClassId handle, char\* methodName,  TriParameterList\* parameterList,  TriParameter\* returnValue) |
| void triRaiseException(in TriExceptionType exc) | void triRaiseException(const TriException\* exc) |

## 6.5 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping

**Clause 8.5.19 TriClassId**

This clause is to be added.

A value of type TriClassIdType represents the name of a class as specified in the TTCN-3 ATS. It is a derived class from QualifiedName, mapped to the following pure virtual class:

class TriClassId : public QualifiedName {

public:

virtual ~TriClassId ();

virtual Tboolean operator== (const TriClassId &sid) const =0;

virtual TriClassId \* cloneClassId () const =0;

virtual Tboolean operator< (const TriClassId &sid) const =0;

}

**Methods:**

* ~TriClassId

Destructor.

* operator==

Returns true if both TriClassId objects are equal.

* cloneClassId

Returns a copy of the TriClassId.

* operator<

Operator < overload.

**Clause 8.5.20 TriObjectHandle**

This clause is to be added.

A value of type TriObjectHandle type is mapped to a void pointer:

typedef void \* TriObjectHandle;

**Clause 8.6.3 TriPlatformPA**

This clause is to be extended.

class TriPlatformPA {

public:

…

//For each constructor on an external class specified in the TTCN-3 ATS implement the behaviour.

virtual TriStatus triExternalCreate (const TriClassId \*classId, TriParameterList \*parameterList, TriObjectHandle \* handle)=0;

//For each destructor on an external class specified in the TTCN-3 ATS implement the behaviour.

virtual TriStatus triExternalCreate (TriObjectHandle handle)=0;

//For each method on an external class specified in the TTCN-3 ATS implement the behaviour.

virtual TriStatus triExternalMethod (TriObjectHandle handle, const Tstring & methodName, TriParameterList \*parameterList, TriParameter \*returnValue)=0;

}

**Clause 8.6.4 TriPlatformTE**

This clause is to be extended.

class TriPlatformTE {

public:

…

//Raises an exception during execution of external code in PA

virtual void triRaiseException (const TriException \*exc)=0;

}

## 6.6 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping

**Clause 9.4.2.19 TriClassId**

This clause is to be added.

**TriClassIdType** C# mapping is derived from the IQualifiedName interface:

public interface ITriClassId : IQualifiedName {}

**Clause 9.4.2.20 TriObjectHandleType mapping**

This clause is to be added.

**TriObjectHandleIdType** is mapped to the C# object class.

**Clause 9.5.2.3 TriPlatformPA**

This clause is to be extended.

public interface ITriPlatformPA {  
 …  
  
 // Miscellaneous operations  
 // Ref: TRI-Definition clause 5.6.3.4  
 TriStatus TriExternalCreate(ITriClassId classId,   
 ITriParameterList parameterList, out object handle);

// Ref: TRI-Definition clause 5.6.3.5  
 TriStatus TriExternalFinally(object handle);

// Ref: TRI-Definition clause 5.6.3.6  
 TriStatus TriExternalMethod(object handle, string methodName,   
 ITriParameterList parameterList, ITriParameter returnValue);  
}

**Clause 9.5.2.4 TriPlatformTE**

This clause is to be extended.

public interface ITriPlatformTE {

…  
  
 // Ref: TRI Definition clause 5.6.3.7

void TriRaiseException(ITriException exc);

}

# 7 TCI Extensions for the Package

## 7.1 Extensions to clause 7.2.2.1 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values

**The definition of the getTypeClass operation shall be modified of the following way:**

TciTypeClassType getTypeClass() Returns the type class of the respective type. A value of TciTypeClassType can have one of the following constants: ADDRESS, ANYTYPE, ARRAY, BITSTRING, BOOLEAN, CHARSTRING, COMPONENT, ENUMERATED, FLOAT, HEXSTRING, INTEGER, OCTETSTRING, RECORD, RECORD\_OF, SET, SET\_OF, UNION, UNIVERSAL\_CHARSTRING, VERDICT, DEFAULT, PORT, TIMER, CLASS.

## 7.2 Extensions to clause 7.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 data types and values

**Clause 7.2.2.5 Abstract TTCN-3 classes**

**This clause is to be added.**

According to the present document, TTCN‑3 user-defined classes will be represented at the TCI interfaces using the abstract data type Class. The abstract data type Class is based on the abstract data type Type.

The following operations defined for abstract data type Type work differently in the abstract data type Class:

Value newInstance() The method creates an ObjectInstance containing a reference to a null object of the class.

The following operations are defined for abstract data type Class:

ObjectInstance create(TriComponentIdType c, TciParameterListType tciPars)  
Calls the constructor to create a new instance of this class using the supplied parameters for the specified component. Created objects are always considered to be initialized.

ClassSeq getSuperclasses () Returns the superclasses of the current class.

TStringSeq getFieldNames () Returns the names of all public fields defined in the class.

TStringSeq getMethodNames () Returns the names of all public methods of the class.

TciParameterTypeListType getConstructorParmeters ()   
Returns formal parameters of the class constructor.

TciParameterTypeListType getMethodParameters (TString methodName)   
Returns formal parameters of the specified public method. The distinct value null is returned if the method does not exist or is not public.

Type getFieldType (TString name) Returns the type of the specified public field. The distinct value null is returned if the member variable does not exist or is not public.

Type getMethodReturnType (TString name)   
Returns the return type of specified public method or the distinct value null if no return value is defined, the method does not exist or it is not public.

**Clause 7.2.2.6 ClassSeq**

**This clause is to be added.**

The abstract data type ClassSeq is used to represent a list of classes.

## 7.3 Extensions to clause 7.2.2.2.0 of ETSI ES 201 873-6 Basic rules

Figure 4 is to be extended.

ObjectInstance

Type

Value

MatchingMechanism

IntegerValue

FloatValue

BooleanValue

CharstringValue

UniversalCharstringValue

BitstringValue

OctetstringValue

HexstringValue

RecordOfValue

RecordValue

UnionValue

EnumeratedValue

VerdictValue

AddressValue

MatchingList

ValueRange

CharacterPattern

MatchDecodedContent

Class

Figure 4: Hierarchy of abstract values

## 7.4 Extensions to clause 7.2.2.2 of ETSI ES 201 873-6 Abstract TTCN-3 values

**Clause 7.2.2.16 The abstract data type ObjectInstance**

This clause is to be added.

The abstract data type ObjectInstance is based on the abstract data type Value. It is used to modify the referenced object and to access public object fields and methods.

The following operations are defined on the abstract data type ObjectInstance:

TriComponentIdType getOwner () Returns the component that owns the object instance.

TString getId () Returns an identifier of the object which is unique within the owner component context.

void setObject (ObjectInstance source)   
The operation sets the referenced object to be the same one as the one referenced by the source parameter. In case the source object does not contain a null reference, the object instance and the source object shall be owned by the same component.

Value getField (TString fieldName)   
Returns the value of the referenced public member field. The distinct value null is returned if the object does not contain the referenced field or the field is not accessible.

Value callMethod(TString methodName, TciParameterListType tciPars)   
Calls the method of the object instance. The distinct value null is returned if the method does not return any value. A runtime error is generated if the method does not exist or if the given parameters do not conform to the formal parameters of the declared method.

## 7.5 Extensions to clause 7.3.4.1 of ETSI ES 201 873-6 Abstract TCI-TL provided

**Clause 7.3.4.1.122 tliObjCreateEnter**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjCreateEnter(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TciParameterListType tciPars) | |
| In Parameters | am | An additional message. |
| ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| c | The component which produces this event. |
| className | The class of the constructor being called. |
| obj | The object being created. |
| tciPars | The parameters of the constructor. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the entering of a constructor of an object. This event occurs after the constructor has been entered. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.123 tliObjCreateLeave**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjCreateLeave(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TciParameterListType tciPars) | |
| In Parameters | am | An additional message. |
| ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| c | The component which produces this event. |
| className | The class of the constructor being called. |
| obj | The created object instance. |
| tciPars | The parameters of the constructor. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the leaving of an object constructor. This event occurs after the constructor has been left. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.124 tliObjFinallyEnter**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjFinallyEnter(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj) | |
| In Parameters | am | An additional message. |
| ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| c | The component which produces this event. |
| className | The class of the finally block being executed. |
| obj | The object instance being destroyed. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the entering of a destructor of an object. This event occurs after the destructor has been entered. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.125 tliObjFinallyLeave**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjCreateLeave(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TciParameterListType tciPars) | |
| In Parameters | am | An additional message. |
| ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| c | The component which produces this event. |
| className | The class of the finally block being executed. |
| obj | The object being destroyed. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the leaving of an object destructor. This event occurs after the destructor has been left. Accessing any members, properties and methods of a destroyed object with exception of methods used for comparison shall cause an error. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.126 tliObjMethodEnter**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjMethodEnter(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TString methodName,  in TciParameterListType tciPars) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| C | The component which produces this event. |
| className | The class of the method being called. |
| obj | The affected object instance. |
| methodName | The name of the called method. |
| tciPars | The parameters of the called method. |
| Return Value | void | |
| Constraint | Shall be called by TE to log the entering of an object method. This event occurs after the method has been entered. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.127 tliObjMethodLeave**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjMethodLeave(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TString methodName,  in TciParameterListType tciPars, in Value returnValue) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| C | The component which produces this event. |
| className | The class of the method being called. |
| obj | The affected object instance. |
| methodName | The name of the called method. |
| tciPars | The parameters of the called method. |
| returnValue | The return value of the called method. |
| Return Value | void | |
| Constraint | Shall be called by TE to log the leaving of an object method. This event occurs after the method has been left. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.132 tliObjVar**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjVar(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c, QualifiedName className,  in ObjectInstance obj, in TString name, in Value value) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| Src | The source file of the test specification. |
| Line | The line number where the request is performed. |
| C | The component which produces this event. |
| className | The class declaring the member variable. |
| obj | The affected object instance. |
| name | The name of the member variable. |
| value | The new value of the member variable. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the modification of the value of a field of an object. This event occurs after the field value has been changed. In case of @lazy fields, it is called also after performing evaluation as the evaluation result is automatically assigned to the field. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.133 tliObjGetEnter**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjGetEnter(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c,  in ObjectInstance obj, in TString propertyName) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| C | The component which produces this event. |
| obj | The affected object instance. |
| propertyName | The name of the referenced property. |
| Return Value | void | |
| Constraint | Shall be called by TE to log the entering of an object getter. This event occurs after the getter has been entered. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.134 tliObjGetLeave**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjGetLeave(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c,  in ObjectInstance obj, in TString propertyName,  in Value value) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| Src | The source file of the test specification. |
| Line | The line number where the request is performed. |
| C | The component which produces this event. |
| Obj | The affected object instance. |
| propertyName | The name of the referenced property. |
| returnValue | The value return by the getter. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the leaving of an object getter. This event occurs after the getter has been left. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.135 tliObjSetEnter**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjSetEnter(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c,  in ObjectInstance obj, in TString propertyName,  in Value value) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| C | The component which produces this event. |
| obj | The affected object instance. |
| propertyName | The name of the referenced property. |
| value | The value passed to the setter. |
| Return Value | Void | |
| Constraint | Shall be called by TE to log the entering of an object setter. This event occurs after the setter has been entered. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

**Clause 7.3.4.1.136 tliObjSetLeave**

This clause is to be added.

|  |  |  |
| --- | --- | --- |
| Signature | void tliObjGetLeave(in TString am, in TInteger ts, in TString src,  in TInteger line, in TriComponentIdType c,  in ObjectInstance obj, in TString propertyName) | |
| In Parameters | Am | An additional message. |
| Ts | The time when the event is produced. |
| src | The source file of the test specification. |
| line | The line number where the request is performed. |
| C | The component which produces this event. |
| obj | The affected object instance. |
| propertyName | The name of the referenced property. |
| Return Value | void | |
| Constraint | Shall be called by TE to log the leaving of an object setter. This event occurs after the setter has been left. | |
| Effect | The TL presents all the information provided in the parameters of this operation to the user, how this is done is not within the scope of the present document. | |

## 7.6 Extensions to clause 8 of ETSI ES 201 873-6 JavaTM language mapping

**Clause 8.3.2.4 TciTypeClassType**

This clause is to be extended.

**TciTypeClassType** is mapped to the following interface:

// TCI IDL TciTypeClassType

package org.etsi.ttcn.tci;

public interface TciTypeClass {

public final static int ADDRESS = 0 ;

public final static int ANYTYPE = 1 ;

public final static int BITSTRING = 2 ;

public final static int BOOLEAN = 3 ;

public final static int CHARSTRING = 5 ;

public final static int COMPONENT = 6 ;

public final static int ENUMERATED = 7 ;

public final static int FLOAT = 8 ;

public final static int HEXSTRING = 9 ;

public final static int INTEGER = 10 ;

public final static int OCTETSTRING = 12 ;

public final static int RECORD = 13 ;

public final static int RECORD\_OF = 14 ;

public final static int ARRAY = 15 ;

public final static int SET = 16 ;

public final static int SET\_OF = 17 ;

public final static int UNION = 18 ;

public final static int UNIVERSAL\_CHARSTRING = 20 ;

public final static int VERDICT = 21 ;

public final static int DEFAULT = 22 ;

public final static int PORT = 23 ;

public final static int TIMER = 24 ;

public final static int CLASS = 25 ;

}

**Clause 8.3.6.7 Abstract class mapping**

This clause is to be added.

**Class** is mapped to the following interface:

// TCI IDL Type

package org.etsi.ttcn.tci;

public interface Class extends Type {

public ObjectInstance create (TriComponentId c, TciParameterList tciPars);

public Class[] getSuperclasses ();

public String[] getFieldNames ();

public String[] getMethodNames ();

public TciParameterTypeList getConstructorParmeters ();

public TciParameterTypeList getMethodParameters (String methodName);

public Type getFieldType (String name);

public Type getMethodReturnType (String methodName);

}

**Methods:**

* create Calls the constructor to create a new instance of this class using the supplied parameters for the specified component.
* getSuperclasses Returns the list of superclasses of this class.
* getFieldNames Returns the names of all public fields defined in the class.
* getMethodNames Returns the names of all public methods of the class.
* getConstructorParmeters Returns the formal parameters of the class constructor.
* getMethodParameters Returns the formal parameters of the specified public method.
* getFieldType Returns the type of the specified public field.
* getMethodReturnType Returns the return type of specified public method or the distinct value null if no return type is declared.

**Clause 8.3.6.8 ClassSeq**

This clause is to be added.

**ClassSeq** abstract data type mapped to an array of TciClass.

**Clause 8.3.4.16 ObjectInstance**

This clause is to be added.

**ObjectInstance** is mapped to the following interface:

// TCI IDL DynamicMatch

package org.etsi.ttcn.tci;

public interface ObjectInstance extends Value {

public TriComponentId getOwner ();

public TString getId ();

public void setObject (ObjectInstance source);

public Value callMethod (String methodName, TciParameterList tciPars);

}

**Methods:**

* getOwner Returns the component that owns the object instance.
* getId Returns the unique identifier of the object instance.
* setObject The operation sets the referenced object to the same reference as the given object.
* getField Gets the value of the referenced public field.
* callMethod Calls a method of the object instance.

**Clause 8.5.4.1 TCI-TL provided**

The TciTLProvided interface is to be extended:

package org.etsi.ttcn.tci;

public interface TciTLProvided {

…

public void tliObjCreateEnter(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, TciParameterList tciPars);

public void tliObjCreateLeave(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, TciParameterList tciPars);

public void tliObjFinallyEnter(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj);

public void tliObjFinallyLeave(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj);

public void tliObjMethodEnter(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String methodName,

TciParameterList tciPars);

public void tliObjMethodLeave(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String methodName,

TciParameterList tciPars, Value returnValue);

public void tliObjVar(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String name, Value value);

public void tliObjGetEnter(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String propertyName);

public void tliObjGetLeave(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String propertyName, Value returnValue);

public void tliObjSetEnter(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String propertyName, Value value);

public void tliObjSetLeave(String am, int ts, String src, int line, TriComponentId c,

QualifiedName className, ObjectInstance obj, String propertyName);

}

## 7.7 Extensions to clause 9 of ETSI ES 201 873-6 ANSI C language mapping

**Clause 9.2 Data**

Table 5 is to be extended.

| TCI IDL Interface | ANSI C representation | Notes and comments |
| --- | --- | --- |
| : |  |  |
| Class | | |
| Value create(TriComponentIdType c, TciParameterListType tciPars) | Value tciObjCreate(Type cls, TriComponentId c, TciParameterListType tciPars) |  |
| ClassSeq getSuperclasses () | Type\* tciGetSuperclasses (Type cls) | Returns null pointer or a null‑pointer terminated array |
| TStringSeq getFieldNames () | String\* tciGetClassFieldNames (Type cls) | Returns null pointer or a null‑pointer terminated array |
| TStringSeq getMethodNames () | String\* tciGetClassMethodNames (Type cls) |
| TciParameterTypeListType getConstructorParmeters () | TciParameterTypeListType\* tciGetClassConstructorParameters (Type cls) |  |
| TciParameterTypeListType getMethodParameters (TString methodName) | TciParameterTypeListType\* tciGetClassMethodParameters (Type cls, String methodName) |  |
| Type getMemberType (TString name) | Type tciGetClassFieldType(Type cls, String name) |  |
| Type getMethodReturnType (TString methodName) | Type tciGetClassMethodReturnType (Type cls, String methodName) |  |
| **ObjectInstance** | | |
| TriComponentIdType getOwner () | TriComponentId tciGetObjOwner (Value obj) |  |
| TString getId () | char \* tciGetObjUniqueId (Value obj) |  |
| void setObject (ObjectInstance source) | void tciSetObject (Value obj, Value source) |  |
| Value getField (TString fieldName) | Value tciGetObjField (Value obj, String fieldName) |  |
| Value callMethod(TString methodName, TciParameterListType tciPars) | Value tciCallObjMethod(Value obj, String methodName, TciParameterListType tciPars) |  |

**Clause 9.4.4.1 TCI-TL provided**

The clause is to be extended.

void tliObjCreateEnter

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

TciParameterListType tciPars);

void tliObjCreateLeave

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

TciParameterListType tciPars);

void tliObjFinallyEnter

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj);

void tliObjFinallyLeave

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj);

void tliObjMethodEnter

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String methodName, TciParameterListType tciPars);

void tliObjMethodLeave

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String methodName, TciParameterListType tciPars, Value returnValue);

void tliObjVar

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String name, Value value);

void tliObjGetEnter

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String propertyName);

void tliObjGetLeave

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String propertyName, Value returnValue);

void tliObjSetEnter

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String propertyName, Value value);

void tliObjSetLeave

(String am, int ts, String src, int line, TriComponentId c, QualifiedName className, Value obj,

String propertyName);

**Clause 9.5 Data**

The definition of the TciTypeClassType in the table 7 is to be modified.

| TCI IDL ADT | ANSI C representation (Type definition) | Notes and comments |
| --- | --- | --- |
| : |  |  |
| TciTypeClassType | typedef enum  {  TCI\_ADDRESS\_TYPE = 0,  TCI\_ANYTYPE\_TYPE = 1,  TCI\_BITSTRING\_TYPE = 2,  TCI\_BOOLEAN\_TYPE = 3,  TCI\_CHARSTRING\_TYPE = 5,  TCI\_COMPONENT\_TYPE = 6,  TCI\_ENUMERATED\_TYPE = 7,  TCI\_FLOAT\_TYPE = 8,  TCI\_HEXSTRING\_TYPE = 9,  TCI\_INTEGER\_TYPE = 10,  TCI\_OCTETSTRING\_TYPE = 12,  TCI\_RECORD\_TYPE = 13,  TCI\_RECORD\_OF\_TYPE = 14,  TCI\_ARRAY\_TYPE = 15,  TCI\_SET\_TYPE = 16,  TCI\_SET\_OF\_TYPE = 17,  TCI\_UNION\_TYPE = 18,  TCI\_UNIVERSAL\_CHARSTRING\_TYPE = 20,  TCI\_VERDICT\_TYPE = 21  TCI\_DEFAULT\_TYPE = 22,  TCI\_PORT\_TYPE = 23,  TCI\_TIMER\_TYPE = 24,  TCI\_CLASS\_TYPE = 25  } TciTypeClassType; |  |
| : |  |  |

## 7.8 Extensions to clause 10 of ETSI ES 201 873-6 C++ language mapping

**Clause 10.5.2.14 TciTypeClass**

This clause is to be extended.

typedef enum

{

TCI\_ADDRESS = 0,

TCI\_ANYTYPE = 1,

TCI\_BITSTRING = 2,

TCI\_BOOLEAN = 3,

TCI\_CHARSTRING = 5,

TCI\_COMPONENT = 6,

TCI\_ENUMERATED = 7,

TCI\_FLOAT = 8,

TCI\_HEXSTRING = 9,

TCI\_INTEGER = 10,

TCI\_OCTETSTRING = 12,

TCI\_RECORD = 13,

TCI\_RECORD\_OF = 14,

TCI\_ARRAY = 15,

TCI\_SET = 16,

TCI\_SET\_OF = 17,

TCI\_UNION = 18,

TCI\_UNIVERSAL\_CHARSTRING = 20,

TCI\_VERDICT = 21

TCI\_DEFAULT = 22,

TCI\_PORT = 23,

TCI\_TIMER = 24

TCI\_CLASS = 25

} TciTypeClass;

**Clause 10.5.3.23 Class**

This clause is to be added.

TTCN-3 class support. It is mapped to the following pure virtual class:

class TciClass : public virtual TciType {

public:

virtual ~TciClass ();

virtual ObjectInstance \* create(const TriComponentId & c, TciParameterList & tciPars) =0;

virtual const std::vector<TciClass\*> & getSuperclasses () const =0;

virtual const std::vector<Tstring\*> & getFieldNames () const =0;

virtual const std::vector<Tstring\*> & getMethodNames () const =0;

virtual const TciParameterTypeList & getConstructorParmeters () const =0;

virtual const TciParameterTypeList & getMethodParameters (Tstring methodName) const =0;

virtual const TciType & getMemberType (const Tstring & name) const =0;

virtual const TciType & getMethodReturnValue (const Tstring & name) const =0;

virtual Tboolean operator== (const TciClass &p\_class) const =0;

virtual TciClass \* clone () const =0;

virtual Tboolean operator< (const TciClass &p\_content) const =0;

}

**Methods:**

~TciClass

Destructor

create

Calls the constructor to create a new instance of this class using the supplied parameters for the specified component

getSuperclasses

Returns the superclasses of the current class

getFieldNames

Returns the names of all public fields defined in the class

getMethodNames

Returns the names of all public methods of the class

getConstructorParmeters

Returns formal parameters of the class constructor

getMethodParameters

Returns formal parameters of the specified public method

getFieldType

Returns the type of the specified public field

getMethodReturnValue

Returns the return type of specified public method or the distinct value null if no return value is defined

operator==

Returns true if both objects are equal

clone

Return a copy of the matching mechanism

operator<

Operator < overload

**Clause 10.5.3.24 ObjectInstance**

This clause is to be added.

TTCN-3 implication and exclusion matching mechanism support. It is mapped to the following pure virtual class:

class ObjectInstance : public virtual TciValue {

public:

virtual ~ObjectInstance ();

virtual const TriComponentId & getOwner () const =0;

virtual const TString getId () const =0;

virtual void setObject (ObjectInstance & val) =0;

virtual TciValue \* getField (const Tstring & fieldName) =0;

virtual Value callMethod(const TString & methodName, TciParameterList & tciPars) =0;

virtual Tboolean operator== (const ObjectInstance &p\_obj) const =0;

virtual ObjectInstance \* clone () const =0;

virtual Tboolean operator< (const ObjectInstance &p\_content) const =0;

}

**Methods:**

~ObjectInstance

Destructor

getOwner

Returns the component that owns the object instance

getId

Returns the unique identifier of the object instance

setObject

The operation sets the referenced object

getField

Returns the value of the referenced public field

callMethod

Calls a method of the object instance

operator==

Returns true if both objects are equal

clone

Return a copy of the matching mechanism

operator<

Operator < overload

**Clause 10.5.3.25 ClassSeq**

This clause is to be added.

The ClassSeq abstract data type is mapped to std::vector<TciClass\*> .

**Clause 10.6.4.1 TciTlProvided**

This clause is to be extended.

//Called by TE to log the entering of a constructor

virtual void tliObjCreateEnter (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj,

const TciParameterList \*tciPars)=0;

//Called by TE to log the leaving of a constructor

virtual void tliObjCreateLeave (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj, const TciParameterList \*tciPars)=0;

//Called by TE to log the entering of a destructor

virtual void tliObjFinallyEnter (const Tstring &am, const timeval ts, const Tstring &src,

const Tinteger line, const TriComponentId \*c, const QualifiedName \*className,

const ObjectInstance \*obj)=0;

//Called by TE to log the leaving of a destructor

virtual void tliObjFinallyLeave (const Tstring &am, const timeval ts, const Tstring &src,

const Tinteger line, const TriComponentId \*c, const QualifiedName \*className,

const ObjectInstance \*obj)=0;

//Called by TE to log the entering of an object method

virtual void tliObjMethodEnter (const Tstring &am, const timeval ts, const Tstring &src,

const Tinteger line, const TriComponentId \*c, const QualifiedName \*className,

const ObjectInstance \*obj, const Tstring &methodName, const TciParameterList \*tciPars)=0;

//Called by TE to log the leaving of an object method

virtual void tliObjMethodLeave (const Tstring &am, const timeval ts, const Tstring &src,

const Tinteger line, const TriComponentId \*c, const QualifiedName \*className,

const ObjectInstance \*obj, const Tstring &methodName, const TciParameterList \*tciPars,

const TciValue \*returnValue)=0;

//Called by TE to log the modification of a member variable of an object

virtual void tliObjVar (const Tstring &am, const timeval ts, const Tstring &src,

const Tinteger line, const TriComponentId \*c, const QualifiedName \*className,

const ObjectInstance \*obj, const Tstring &name, const TciValue \*value)=0;

//Called by TE to log the entering of an object getter

virtual void tliObjGetEnter (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj,

const Tstring &propertyName)=0;

//Called by TE to log the leaving of an object getter

virtual void tliObjGetLeave (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj,

const Tstring &propertyName, const TciValue \*returnValue)=0;

//Called by TE to log the entering of an object setter

virtual void tliObjSetEnter (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj,

const Tstring &propertyName, const TciValue \*value)=0;

//Called by TE to log the leaving of an object setter

virtual void tliObjSetEnter (const Tstring &am, const timeval ts, const Tstring &src, const Tinteger line, const TriComponentId \*c, const QualifiedName \*className, const ObjectInstance \*obj,

const Tstring &propertyName)=0;

## 7.9 Extensions to clause 11 of ETSI ES 201 873-6 W3C XML mapping

**Clause 11.3.3.30 ObjectInstance**

ObjectInstance type is mapped to the complex type specified below. The content of the XML elements based on the ObjectInstance type shall be equal to the string produced by the valueToString operation (described in clause 7.2.2.2.1 of ETSI ES 201 873-6 [4]):

<xsd:complexType name="ObjectInstance">

<xsd:group ref="Values:BaseValue"/>

<xsd:attributeGroup ref="Values:ValueAtts"/>

</xsd:complexType>

**Items:**

* BaseValue Object instance content described in clause 11.3.3.1 of ETSI ES 201-873-6
* ValueAtts Value attributes described in clause 11.3.3.1 of ETSI ES 201-873-6

**Clause 11.4.2.1 TCI-TL provided**

**This clause is to be extended.**

<xsd:complexType name="tliObjCreateEnter">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjCreateLeave">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>

<xsd:element name="returnValue" type="Values:Value" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjFinallyEnter">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjFinallyLeave">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjMethodEnter">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="methodName" type="SimpleTypes:TString" />

<xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjMethodLeave">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="methodName" type="SimpleTypes:TString" />

<xsd:element name="tciPars" type="Types:TciParameterListType" minOccurs="0"/>

<xsd:element name="returnValue" type="Values:Value" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjVar">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="name" type="SimpleTypes:TString" />

<xsd:element name="val" type="Values:Value" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

xsd:complexType name="tliObjGetEnter">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="propertyName" type="SimpleTypes:TString" />

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjGetLeave">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="propertyName" type="SimpleTypes:TString" />

<xsd:element name="returnValue" type="Values:Value" minOccurs="0"/>

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

<xsd:complexType name="tliObjSetEnter">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="propertyName" type="SimpleTypes:TString" />

<xsd:element name="value" type="Values:Value" />

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

xsd:complexType name="tliObjSetLeave">

<xsd:complexContent mixed="true">

<xsd:extension base="Events:Event">

<xsd:sequence>

<xsd:element name="className" type="Types:QualifiedName" />

<xsd:element name="obj" type="Values:ObjectInstance" />

<xsd:element name="propertyName" type="SimpleTypes:TString" />

</xsd:sequence>

</xsd:extension>

</xsd:complexContent>

</xsd:complexType>

## 7.10 Extensions to clause 12 of ETSI ES 201 873-6 C# language mapping

**Clause 12.4.2.4 TciTypeClassType**

This clause is to be extended.

**TciTypeClassType** is mapped to the following enumeration:

public enum TciTypeClass {  
 Address = 0,  
 Anytype = 1,  
 Bitstring = 2,   
 BooleanType = 3,  
 Charstring = 5,  
 Component = 6,  
 Enumerated = 7,  
 Float = 8,  
 Hexstring = 9,  
 IntegerType = 10,   
 Octetstring = 12,  
 Record = 13,  
 RecordOf = 14,  
 Array = 15  
 Set = 16,  
 SetOf = 17,  
 Union = 18,  
 UniversalCharstring = 20,  
 Verdict = 21,  
 Default = 22,  
 Port = 23,  
 Timer = 24,  
 Class = 25  
}

**Clause 12.4.7 Abstract class mapping**

This clause is to be added.

The IDL type **Class** is mapped to the following interface:

// TCI IDL Type

package org.etsi.ttcn.tci;

public interface ITciClass : ITciType {

ITciObjectInstance Create (ITriComponentId c, ITciParameterList tciPars);

ITciClass[] Superclasses { get; }

String[] FieldNames { get };

String[] MethodNames { get; }

ITciParameterTypeList ConstructorParmeters { get; }

ITciParameterTypeList GetMethodParameters (String methodName);

ITciType GetFieldType (String name);

ITciType GetMethodReturnType (String methodName);

}

**Methods:**

* Create Calls the constructor to create a new instance of this class using the supplied parameters for the specified component.
* Superclasses Returns the list of superclass of the current.
* FieldNames Returns the names of all public fields defined in the class.
* MethodNames Returns the names of all public methods of the class.
* ConstructorParmeters Returns formal parameters of the class constructor.
* GetMethodParameters Returns formal parameters of the specified public method.
* GetFieldType Returns the type of the specified public field.
* GetMethodReturnType Returns the return type of specified public method or the distinct value null if no return value is defined.

**Clause 12.4.8 ClassSeq mapping**

This clause is to be added.

The **ClassSeq** abstract data type is mapped to **ITciClass[]**.

**Clause 12.4.4.16 ObjectInstance**

This clause is to be added.

The IDL type **ObjectInstance** is mapped to the following interface:

public interface ITciObjectInstance : ITciValue {

ITciComponentId Owner { get; }

String Id { get; }

void SetObject (ITciObjectInstance source);

ITciValue GetField (String fieldName);

ITciValue CallMethod (String methodName, ITciParameterList tciPars);

}

**Methods:**

* Owner Returns the component that owns the object instance.
* Id Returns the unique identifier of the object instance.
* SetObject The operation sets the referenced object.
* GetField Returns the value of the referenced public field.
* CallMethod Calls a method of the object instance.

**Clause 12.5.4.1 TCI-TL provided**

The ITciTLProvided interface is to be extended:

public interface ITciTLProvided {

…

void TliObjCreateEnter(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj,

ITciParameterList tciPars);

void TliObjCreateLeave(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj,

ITciParameterList tciPars);  
 void TliObjFinallyEnter(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj);

void TliObjFinallyLeave(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj);

void TliObjMethodEnter(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string methodName,

ITciParameterList tciPars);  
 void TliObjMethodLeave(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string methodName,

ITciParameterList tciPars, ITciValue returnValue);  
 void TliObjVar (string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string name,

ITciValue value);

void TliObjGetEnter(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string propertyName);

void TliObjGetLeave(string am, System.DateTime ts, string src, int line,  
 ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string propertyName,

ITciValue returnValue);

void TliObjSetEnter(string am, System.DateTime ts, string src, int line,

ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string propertyName,

ITciValue value);  
 void TliObjSetLeave(string am, System.DateTime ts, string src, int line,  
 ITriComponentId c, IQualifiedName className, ITciObjectInstance obj, string propertyName);  
}

# 8 XTRI Extensions for the Package (optional)

## 8.1 Changes to clause 5.6.3 of ETSI ES 201 873-5 Miscellaneous operations

**Clause 5.6.3.4 triExternalCreate 🡪 xtriExternalCreate**

|  |  |
| --- | --- |
| **Signature** | TriStatusType xtriExternalCreate(  in TriClassIdType classId,  inout TciParameterListType parameterList, out TriObjHandleType createdObject) |
| In Parameters | classId identifier of the external class |
| **Out Parameters** | createdObject handle to the created object |
| **InOutParameters** | parameterList a list of ~~encoded~~ parameters for the indicated constructor. The parameters in parameterList are ordered as they appear in the TTCN‑3 constructor declaration. |
| **Return Value** | The return status of the xtriExternalCreate operation. The return status indicates the local success (***TRI\_OK***) or failure (***TRI\_Error***) of the operation. |
| **Constraints** | This operation is called by the TE when it invokes a constructor specified in a class which is defined as external in TTCN‑3.  In the invocation of a triExternalCreate operation by the TE all *in* and *inout* constructor parameters contain ~~encoded~~ values. No error shall be indicated by the PA in case the value of any *out* parameter is non‑null. |
| **Effect** | The PA shall implement the behaviour for each external class specified in the TTCN‑3 ATS. On invocation of this operation, the PA shall invoke the constructor of a class indicated by the identifier classId. It shall access the specified *in* and *inout* constructor parameters in parameterList, create a new external object instance using the values of these parameters, and compute values for *inout* and *out* parameters in parameterList. The operation shall then return ~~encoded~~ values for all *inout* and *out* constructor parameters and a handle to the created external object.  The xtriExternalCreate operation returns ***TRI\_OK*** if the PA completes the constructor of the external class successfully, ***TRI\_Error*** otherwise. In the latter case, the distinct value null shall be returned as the object handle.  Note that whereas most of other TRI operations are considered to be non‑blocking, the xtriExternalCreate operation is considered to be *blocking.* That means that the operation shall not return before the construction of the external object has been finished. External constructors have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. |

**Clause 5.6.3.6 triExternalMethod 🡪 xtriExternalMethod**

|  |  |
| --- | --- |
| **Signature** | TriStatusType xtriExternalMethod(  in TriObjHandleType handle,  in String methodName,  inout TciParameterListType parameterList, out TciParameterType returnValue) |
| **In Parameters** | handle handle of the affected object; null for static methods  methodName name of the called method |
| **Out Parameters** | returnValue (optional) ~~encoded~~ return value |
| **InOutParameters** | parameterList a list of ~~encoded~~ parameters for the indicated method. The parameters in parameterList are ordered as they appear in the TTCN‑3 method declaration. |
| **Return Value** | The return status of the xtriExternalMethod operation. The return status indicates the local success (***TRI\_OK***) or failure (***TRI\_Error***) of the operation. |
| **Constraints** | This operation is called by the TE when it invokes a method specified in a class which is defined as external in TTCN‑3.  In the invocation of a xtriExternalMethod operation by the TE all *in* and *inout* parameters contain ~~encoded~~ values. No error shall be indicated by the PA in case the value of any *out* parameter is non‑null. |
| **Effect** | The PA shall implement the behaviour for each method of all external classes specified in the TTCN‑3 ATS. On invocation of this operation, the PA shall call a method methodName of an external object whose handle is in the handle parameter. It shall access the specified *in* and *inout* method parameters in parameterList, pass the values of these parameters to the called method, and compute values for *inout* and *out* parameters in parameterList. The operation shall then return ~~encoded~~ values for all *inout* and *out* method parameters and the ~~encoded~~ return value.  If no return type has been defined for this method in the TTCN‑3 ATS, the distinct value null shall be used for the latter.  The xtriExternalMethod operation returns ***TRI\_OK*** if the PA completes the method of the external object successfully, ***TRI\_Error*** otherwise.  Note that whereas most of other TRI operations are considered to be non‑blocking, the xtriExternalMethod operation is considered to be *blocking.* That means that the operation shall not return before the method call has been finished. Methods of external classes have to be implemented carefully as they could cause deadlock of test component execution or even the entire test system implementation. |

**Clause 5.6.3.7 triRaiseException 🡪 xtriRaiseException**

|  |  |
| --- | --- |
| **Signature** | void xtriRaiseException(in Value exc) |
| **In Parameters** | exc ~~encoded~~ exception to raise |
| **Constraints** | This operation can be called by the PA during execution of triExternalFunction, triExternalCreate, triExternalFinally or triExternalMethod. |
| **Effect** | The operation raises an exception that can be later processed by the TE in the catch statement. ~~The exception is provided in an enoded form.~~ ~~Decoding is performed in the catch statement.~~ |

## 8.2 Extensions to clause 6 of ETSI ES 201 873-5 JavaTM language mapping

**Clause 6.5.3.1 Changes to TriPlatformPA**

This clause is to be extended.

// TriPlatform

// TE ‑> PA

package org.etsi.ttcn.tri;

public interface xTriPlatformPA {

…

// Ref: TRI‑Definition 5.6.3.4

public TriStatus triExternalCreate(TriClassIdType classId,

TciParameterList parameterList, TriObjHandleWrapper handle);

// Ref: TRI‑Definition 5.6.3.6

public TriStatus xtriExternalMethod(Object handle, String methodName,

TciParameterList parameterList, TciParameter returnValue);

}

**Clause 6.5.3.1 Changes to TriPlatformPA**

This clause is to be extended.

// TriPlatform

// PA ‑> TE

package org.etsi.ttcn.tri;

public interface xTriPlatformTE {

…

// Ref: TRI-Definition 5.6.3.7

public void triRaiseException(Value exc);

}

## 8.3 Extensions to clause 7 of ETSI ES 201 873-5 ANSI C language mapping

**Clause 7.2.4 TRI operation mapping**

This clause is to be extended.

| IDL Representation | ANSI C Representation |
| --- | --- |
| **…** |  |
| TriStatusType xtriExternalCreate (in TriClassIdType classId, inout TciParameterListType parameterList, out TriObjHandleType createdObject) | TriStatus xtriExternalCreate  (const TriClassId\* classId,  TciParameterList\* parameterList,  TriObjectHandle\* handle) |
| TriStatusType xtriExternalMethod(in TriObjHandleType handle, in String methodName, inout TciParameterListType parameterList, out TciParameterType returnValue) | TriStatus xtriExternalFunction  (TriClassId handle, char\* methodName,  TciParameterList\* parameterList,  TciParameter\* returnValue) |
| void xtriRaiseException(in Value exc) | void xtriRaiseException(const Value\* exc) |

## 8.4 Extensions to clause 8 of ETSI ES 201 873-5 C++ language mapping

**Clause 8.6.3 TriPlatformPA**

This clause is to be extended.

class xTriPlatformPA {

public:

…

//For each constructor on an external class specified in the TTCN-3 ATS implement the behaviour.

virtual TriStatus xtriExternalCreate (const TriClassId \*classId, TciParameterList \*parameterList, TriObjectHandle \* handle)=0;

//For each method on an external class specified in the TTCN-3 ATS implement the behaviour.

virtual TriStatus xtriExternalMethod (TriObjectHandle handle, const Tstring & methodName, TciParameterList \*parameterList, TciParameter \*returnValue)=0;

}

**Clause 8.6.4 TriPlatformTE**

This clause is to be extended.

class xTriPlatformTE {

public:

…

//Raises an exception during execution of external code in PA

virtual void xtriRaiseException (const TciValue \*exc)=0;

}

## 8.5 Extensions to clause 9 of ETSI ES 201 873-5 C# language mapping

**Clause 9.5.2.3 TriPlatformPA**

This clause is to be extended.

public interface IXTriPlatformPA {  
 …  
  
 // Miscellaneous operations  
 // Ref: TRI-Definition clause 5.6.3.4  
 TriStatus XTriExternalCreate(ITriClassId classId,   
 ITciParameterList parameterList, out object handle);

// Ref: TRI-Definition clause 5.6.3.6  
 TriStatus XTriExternalMethod(object handle, string methodName,   
 ITciParameterList parameterList, ITciParameter returnValue);  
}

**Clause 9.5.2.4 TriPlatformTE**

This clause is to be extended.

public interface IXTriPlatformTE {

…  
  
 // Ref: TRI Definition clause 5.6.3.7

void XTriRaiseException(ITciValue exc);

}

Annex A (normative):  
BNF and static semantics

# A.1 Extensions to TTCN-3 terminals

The list of reserved terminals which are keywords in table A.3 in ETSI ES 201 873-1 [1] needs to be extended with **class**, **finally, object** and **this**. The extension of table A.3 in ETSI ES 201 873-1 [1], clause A.1.5.0 is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| **action**  **activate**  **address**  **alive**  **all**  **alt**  **altstep**  **and**  **and4b**  **any**  **anytype**  **bitstring**  **boolean**  **break**  **case**  **call**  **catch**  **char**  **charstring**  **check**  **class**  **clear**  **complement**  **component**  **connect**  **const**  **continue**  **control**  **create**  **deactivate**  **decmatch**  **default**  **disconnect**  **display**  **do**  **done**  **else**  **encode**  **enumerated**  **error**  **except**  **exception**  **execute**  **extends**  **extension**  **external** | **fail**  **false**  **finally**  **float**  **for**  **friend**  **from**  **function**  **getverdict**  **getcall**  **getreply**  **goto**  **group**  **halt**  **hexstring**  **if**  **ifpresent**  **import**  **in**  **inconc**  **infinity**  **inout**  **integer**  **interleave**  **kill**  **killed**  **label**  **language**  **length**  **log**  **map**  **match**  **message**  **mixed**  **mod**  **modifies**  **module**  **modulepar**  **mtc** | **noblock**  **none**  **not**  **not4b**  **nowait**  **null**  **object**  **octetstring**  **of**  **omit**  **on**  **optional**  **or**  **or4b**  **out**  **override**  **param**  **pass**  **pattern**  **permutation**  **port**  **present**  **private**  **procedure**  **public**  **raise**  **read**  **receive**  **record**  **recursive**  **rem**  **repeat**  **reply**  **return**  **running**  **runs** | **select**  **self**  **send**  **sender**  **set**  **setverdict**  **signature**  **start**  **stop**  **subset**  **superset**  **system**  **template**  **testcase**  **this**  **timeout**  **timer**  **to**  **trigger**  **true**  **type**  **union**  **universal**  **unmap**  **value**  **valueof**  **var**  **variant**  **verdicttype**  **while**  **with**  **xor**  **xor4b** |

# A.2 Modified TTCN-3 syntax BNF productions

This clause includes all BNF productions that are modifications of BNF rules defined in the TTCN-3 core language document ETSI ES 201 873-1 [1]. When using this package the BNF rules below replace the corresponding BNF rules in the TTCN-3 core language document. The rule numbers define the correspondence of BNF rules.

Additions to clause A.1.6 of ETSI ES 201 873-1 are identified by underlined font, deletions are identified by strikethrough font. In case of contradiction between the above clause of part 1 and this clause (i.e. parts of the productions not marked by ~~strikethrough~~ font is changed in part 1), part 1 takes precedence, i.e. tools supporting the present document shall apply the insertions and deletions of this clause to the actual part 1 production or static semantics rule automtically.

BNF changes in clause A.1.6.1.1 Type definitions

12. StructuredTypeDef ::= RecordDef |

UnionDef |

SetDef |

RecordOfDef |

SetOfDef |

EnumDef |

PortDef |

ComponentDef |

MapDef |

ClassDef

BNF Changes in clause A.1.6.1.3 Template Definitions

100.FieldSpec ::= ( FieldReference | FunctionInstance ) AssignmentChar (TemplateBody | Minus)

BNF changes in clause A.1.6.1.4 Function definitions

161. FunctionDef ::= FunctionKeyword [ DeterministicModifier | ControlModifier ]

IdentifierOrControl "(" [ FunctionFormalParList ] ")"

[ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ] [ ReturnType ] [ ExceptionSpec ]

StatementBlock

172. StatementBlock ::= BasicStatementBlock [ CatchBlocks ] [ FinallyBlock ]

179. FunctionRef ::= [ ( Identifier | ObjectInstance ) Dot ]

( Identifier | PreDefFunctionIdentifier | ControlKeyword )

BNF changes in clause A.1.6.1.7 Altstep definitions

195. AltstepDef ::= AltstepKeyword [ ControlModifier ] [ InterleaveKeyword ]

Identifier "(" [ FunctionFormalParList ] ")"

[ RunsOnSpec ] [ MtcSpec ] [ SystemSpec ] [ ExceptionSpec ]

"{" AltstepLocalDefList AltGuardList "}" [ CatchBlocks ] [ FinallyBlock ]

BNF changes in clause A.1.6.1.10 External function definitions

247. ExtFunctionDef ::= ExtKeyword FunctionKeyword [ DeterministicModifier | ControlModifier ]

Identifier "(" [ FunctionFormalParList ] ")"

[ ReturnType ] [ ExceptionSpec ]

BNF changes in clause A.1.6.3.1 Variable instantiation

259.SingleVarInstance ::= [PropertyModifier] Identifier [ArrayDef]

[AssignmentChar Expression] [PropertyBody]

/\* STATIC PropertyBody shall only be present if PropertyModifier is also present \*/

262.SingleTempVarInstance ::= [PropertyModifier] Identifier [ArrayDef]

[AssignmentChar TemplateBody] [PropertyBody]  
/\* STATIC PropertyBody shall only be present if PropertyModifier is also present \*/

263. ValueRef ::= [ ThisOp Dot ] Identifier [ExtendedFieldReference]

BNF changes in clause A.1.6.4.1 Component Operations

269. CreateOp ::= Type Dot CreateKeyword [ActualParList]

[AliveKeyword |

ExternalKeyword ActualParList]

BNF changes in clause A.1.6.5 Type

405. PredefinedType ::= BitStringKeyword |

BooleanKeyword |

CharStringKeyword |

UniversalCharString |

IntegerKeyword |

OctetStringKeyword |

HexStringKeyword |

VerdictTypeKeyword |

FloatKeyword |

AddressKeyword |

DefaultKeyword |

AnyTypeKeyword |

TimerKeyword |

ObjectKeyword

BNF changes in clause A.1.6.6 Value

437. ReferencedValue ::= ( ( ExtendedIdentifier | ThisOp ) [ExtendedFieldReference] )

| ReferencedEnumValue

BNF changes in clause A.1.6.8.2 Behaviour statements

492. BehaviourStatements ::= TestcaseInstance |

FunctionInstance |

ReturnStatement |

AltConstruct |

InterleavedConstruct |

LabelStatement |

GotoStatement |

RepeatStatement |

DeactivateStatement |

AltstepInstance |

ActivateOp |

BreakStatement |

ContinueStatement |

RaiseExceptionStatement

BNF changes in clause A.1.6.8.3 Basic statements

562. RelOp ::= "<" | ">" | ">=" | "<=" | OfKeyword

# 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 "033". The numbering of other rules start with number 033001.

Additional BNF rules related to clause A.1.6.1.1 Type definitions

033001. ClassDef ::= [ ExtKeyword ] ClassKeyword [ FinalModifier | AbstractModifier | TraitModifier]

Identifier [ ExtendsKeyword ClassTypeList ] [ RunsOnSpec ] [ MtcSpec ]

[ SystemSpec ]

"{" ClassMemberList "}"

[ FinallyKeyword BasicStatementBlock ]

033002. ClassKeyword ::= "class"

033003. ThisOp ::= "this"

033004. SuperOp ::= "super"

033005. FinalModifier ::= "@final"

033006. AbstractModifier ::= "@abstract"

033007. FinallyKeyword ::= "finally"

033008. ObjectKeyword ::= "object"

033008a. ClassType ::= ReferencedType | ObjectKeyword

/\* STATIC SEMANTICS – ReferencedType shall evaluate to a class. \*/

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

033010. ClassMember ::= [ MemberVisibility ]

( VarInstance |

TimerInstance |

ClassConstDef |

ClassTemplateDef |

ClassFunctionDef |

ConstructorDef |

ClassDef )

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

033012. ClassFunctionDef ::= [ ExtKeyword ] FunctionKeyword

[ FinalModifier | AbstractModifier ] [ DeterministicModifier ]

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

[ StatementBlock ]

033013. ConstructorDef ::= CreateKeyword

"(" FunctionFormalParList ")"

[ ExternalKeyword "(" FunctionFormalParList ")" ]

[ ":" ReferencedType ActualParList ]

[ StatementBlock ]

/\* STATIC SEMANTICS – ReferencedType shall evaluate to a class. \*/

033013a. ClassConstDef ::= ConstKeyword Type ClassConstList

033013a ClassConstList ::= SingleClassConstDef {"," SingleClassConstDef}

033013a SingleClassConstDef ::= Identifier [ArrayDef] [ AssignmentChar ConstantExpression ]

033013b. ClassTemplateDef ::= TemplateKeyword [TemplateRestriction]

[FuzzyModifier [DeterministicModifier]]

BaseTemplate [DerivedDef] [ AssignmentChar BaseTemplateBody ]

033014. TraitModifier ::= "@trait"

003015. ClassTypeList ::= ClassType { "," ClassType }

Additional BNF rules related to clause A.1.6.1.4 Function definitions

033016. BasicStatementBlock ::= "{" [ FunctionDefList ] [ FunctionStatementList ] "}"

033017. CatchBlocks ::= CatchBlock { CatchBlock }

033018. CatchBlock ::= CatchOpKeyword "(" Type Identifier ")" BasicStatementBlock

033019. FinallyBlock ::= FinallyKeyword BasicStatementBlock

033020. ObjectInstance ::= ( ThisOp | ValueRef | FunctionInstance ) [ ExtendedFieldReference ]

Additional BNF related to clause A.1.6.3.1 Variable Instantiation

033021. PropertyModifier ::= "@property"

033022. PropertyBody ::= "{" ( PropertyGetterDef [PropertySetterDef] |

PropertySetterDef [PropertyGetterDef] ) "}"

033023. PropertyGetterDef ::= ["public"| "private"]

[AbstractModifier | FinalModifier] [DeterministicModifier]

"@get" [ "=>" TemplateBody | *StatementBlock* ] [";"]

033024. PropertySetterDef ::= ["public"| "private"]

[AbstractModifier | FinalModifier] [DeterministicModifier]

"@set" [ "=>" Assignment | StatementBlock] [";"]

Additional BNF rules related to clause A.1.6.8.2 Behaviour statements

033025. RaiseExceptionStatement ::= RaiseKeyword TemplateInstance

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

Annex B (normative):  
Standard Collections

# B.1 The TTCN3\_standard\_collections module

## B.1.0 General

The classes and external functions defined in this module provide users with the following commonly used data structures.

**module** TTCN3\_standard\_collections {

**function** instanceEqual(object element1, object element2) **return boolean** {   
 **return** element1 == element2   
}

**public** **external function** createLinkedList(**in** equalsFunctionType equalsFunction := instanceEqual) **return** LinkedList;

**public** **external function** createQueue(**in** equalsFunctionType equalsFunction := instanceEqual)   
**return** Queue;

**public external function** createPriorityQueue(**in** comparatorFunctionType comparatorFunction)   
**return** PriorityQueue;

**public external function** createStack(**in** equalsFunctionType equalsFunction := instanceEqual)   
**return** Stack;

**public external function** createRingBuffer(**in integer** maxSize) **return** RingBuffer;

**public external function** createHashMap(**in** hashFunctionType hashFunction,   
 **in** equalsFunctionType equalsFunction) **return** HashMap;

**public external function** createSet(**in** equalsFunctionType equalsFunction := instanceEqual)   
**return** Set;

**public type class @abstract** Exception {

}

**type class @abstract** Iterator {

**function @abstract** hasNext() **return boolean**;

**function @abstract** next() **return object**;

}

**type class @abstract** Collection {

**function** size() **return** **integer**;

**function** contains(**object** element) **exception** Exception **return** **boolean**;

**function** **@abstract** iterator() **return** Iterator;

}

**type class @abstract** List **extends** Collection {

**public function @abstract** add(**object** element) **exception** Exception;

**public function @abstract** remove(**object** element) **exception** Exception **return** **boolean**;

**public function @abstract** get(**integer** index) **exception** Exception **return** **object**;

}

**public type class @abstract** LinkedList **extends** List {

**public function @abstract** getFirst() **exception** Exception **return object**;

**public function @abstract** getLast() **exception** Exception **return object**;

**public function @abstract** removeFirst() **exception** Exception **return object**;

**public function @abstract** removeLast() **exception** Exception **return object**;

**public function @abstract** addFirst(**object** element) **exception** Exception;

**public function @abstract** addLast(**object** element) **exception** Exception;

}

**public type class @abstract** Queue extends Collection {

**public function @abstract** add(**object** element) **exception** Exception;

**public function @abstract** remove() **exception** Exception **return object**;

}

**public type function** comparatorFunctionType(**in object** element1, **in object** element2)  
**exception** Exception **return integer**;

**public type class @abstract** PriorityQueue **extends** Queue {

}

**public type class @abstract** Stack **extends** Collection {

**public function @abstract** push(**object** element) **exception** Exception;

**public function @abstract** pop() **exception** Exception **return object**;

}

**public type class @abstract** RingBuffer **extends** Collection {

**public function @abstract** put(**object** element) **exception** Exception;

**public function @abstract** get() **exception** Exception **return object**;

**public function @abstract** capacity() **return integer**;

}

**public type function** hashFunctionType(**in object** element) **exception** Exception **return integer**;

**public type function** equalsFunctionType(**in object** element1, **in object** element2) **exception** Exception **return boolean**;

**public type class** **@abstract** HashMap {

**public function @abstract** put(**object** keyElement, **object** valueElement) **exception** Exception;

**public function @abstract** get(**object** keyElement) **exception** Exception **return object**;

**public function @abstract** containsKey(**object** keyElement) **exception** Exception **return boolean**;

**public function @abstract** remove(**object** keyElement) **exception** Exception **return object**;

**public function @abstract** keySet() **return Set**;

**public function @abstract** values() **return** List;

**public function @abstract** size() **return integer**;

}

**public type class @abstract** Set **extends** Collection {

**public function @abstract** add(**object** element) **exception** Exception **return** **boolean**;

**public function @abstract** remove(**object** element) **exception** Exception;

}

}

## B.1.1 The Collection class

The abstract [Collection](#Collection) class represents a data structure that is a collection of elements. It is used as a base class of more specific collection data structures like lists and sets.

External function and class methods:

* size  
  Returns the number of elements stored in the LinkedList.
* contains  
  Returns the value true if the given element is contained at least once in the collection.
* iterator  
  Returns an Iterator object for iterating over the elements of the collection.

## B.1.2 The List class

The abstract [List](#List) class represents a list of elements where each contained element has an index (starting from 0).

External function and class methods:

* add  
  Adds the given element to the list.
* remove  
  Tries to remove one instance of the provided element from the List.  
  Returns **true** if an element was removed, **false** if no elements were removed.  
  Please note, that a List might contain the same element several times, in which case only one instance will be removed.
* get  
  Gets the element at the given index from the list, if the index in in the range (0 .. size()-1).

## B.1.3 The LinkedList class

The abstract [LinkedList](#LinkedList) class represents a double linked data structure for storing objects.  
A new Instance can be created via the external function **createLinkedList**. It is derived from the abstract List class.

External function and class methods:

* createLinkedList  
  Factory function for creating a new LinkedList instance.
* getFirst  
  Returns the first element of the LinkedList if it is not empty.  
  Raises an exception if the LinkedList is empty.
* getLast  
  Returns the last element of the LinkedList if it is not empty.  
  Raises an exception if the LinkedList is empty.
* removeLast  
  Removes and returns the last element of the LinkedList if it is not empty.  
  Raises an exception if the LinkedList is empty.
* addFirst  
  Adds a new element as the first element of the LinkedList if this is possible.  
  Raises an exception in case of error, for example: running out of memory.
* addLast  
  Adds a new element as the last element of the LinkedList if this is possible.  
  Raises an exception in case of error, for example: running out of memory.
* iterator  
  Returns an iterator over the elements of this LinkedList.  
  The elements are iterated from first to last.
* size  
  Returns the number of elements stored in the LinkedList.

## B.1.4 The Queue class

The abstract [Queue](#Queue) class represents a queue data structure for storing objects. This data structure uses a First In First Out semantics, meaning that the element added first will also be the element removed first. It is derived from the abstract class Collection.  
A new Instance can be created via the external function **createQueue**.

External function and class methods:

* createQueue  
  Factory function for creating a new Queue instance.
* add  
  Adds an element to the end of the Queue if this is possible.  
  Raises an exception in case of error, for example: running out of memory.
* remove  
  Removes and returns the first element of the Queue if it is not empty.  
  Raises an exception if the Queue is empty.
* size  
  Returns the number of elements stored in the Queue.

## B.1.5 The PriorityQueue class

The abstract [PriorityQueue](#PriorityQueue) class represents a priority queue data structure for storing objects. This data structure stores its elements ordered according to the comparator function. This data structure does not allow null elements.  
A new Instance can be created via the external function **createPriorityQueue**. It is derived from the class Queue.

External function and class methods:

* createPriorityQueue  
  Factory function for creating a new PriorityQueue instance.
* comparatorFunctionType  
  Used to compare the 2 provided elements for their ordering.  
  Returns a negative integer if the element1 is less than element2.  
  Returns 0 if the element1 is equivalent to element2.  
  Return a positive integer if element1 is greater than element2.  
  Functions of this type can also raise an exception, for example if the object received as one of their actual parameters is not of the expected class.
* add  
  Adds an element to the PriorityQueue if this is possible. The element will be added before all elements that are greater than the element and after all elements that are smaller than or equal to the element. Thereby it is ensured that always the smallest element first added to the queue is at the head of the queue.   
  Raises an exception in case of error, for example: running out of memory or adding a null object.
* remove  
  Removes and returns the head element of the PriorityQueue if it is not empty. The head element has the lowest priority among the elements of the PriorityQueue.  
  Raises an exception if the PriorityQueue is empty.
* size  
  Returns the number of elements stored in the PriorityQueue.

## B.1.6 The Stack class

The abstract [Stack](#Stack) class represents a stack data structure for storing objects. This data structure uses a Last In First Out semantics, meaning that the element added last will also be the element removed first.  
A new Instance can be created via the external function **createStack**.

External function and class methods:

* createStack  
  Factory function for creating a new Stack instance.
* push  
  Pushes an element onto the Stack if this is possible.  
  Raises an exception in case of error, for example: running out of memory.
* pop  
  Removes and returns the element inserted last from the Stack if it is not empty.  
  Raises an exception if the Stack is empty.
* size  
  Returns the number of elements stored in the Stack.

## B.1.7 The RingBuffer class

The abstract [RingBuffer](#RingBuffer) class represents a ringbuffer data structure for storing objects. This data structure uses a First In First Out semantics, with a fixed size limit. This means that the element added first will also be the element removed first. An instance of this collection can only accept elements to the maximum amount it is created for.  
A new Instance can be created via the external function **createRingBuffer**.

External function and class methods:

* createRingBuffer  
  Factory function for creating a new RingBuffer instance, with the provided maximum size.
* put  
  Adds an element to the end of the RingBuffer if this is possible.  
  Raises an exception in case of error, for example: reaching the maximum allowed size of the buffer.
* get  
  Removes and returns the first element of the RingBuffer if it is not empty.  
  Raises an exception if the RingBuffer is empty.
* size  
  Returns the number of elements stored in the RingBuffer.
* capacity  
  Returns the maximum capacity of the RingBuffer.

## B.1.8 The HashMap class

The abstract [HashMap](#HashMap) class represents a hashmap data structure for storing key-value pairs of objects. This collection can be indexed with the keyElement part of the pair, to receive the valueElement of the pair.  
Please note that each key has to be unique according to the given **equalsFunction**.

A new Instance can be created via the external function **createHashMap**.  
The hash value of the keyElement object can be calculated using the provided **hashFunctionType** function and the equality of two given keyElements can be determined using the provided **equalsFunctionType** function.

External function and class methods:

* createHashMap  
  Factory function for creating a new HashMap instance, that will use the provided hashFunction for calculating the hash values of the key element objects and an equalsFunction for determining the equality of keys. The two functions need to fulfil the property that for all pairs of objects o1, o2, if equalsFunction(o1,o2) is true then also hashFunction(o1)==hashFunction(o2) is true.
* hashFunctionType  
  A behaviour type allowing the user of the collection to provide their implementation for calculating the hash value of their key element objects.  
  Functions of this type will be called with a key element object as their only parameter and shall return an integer value that represents the hash value of the object.  
  Functions of this type can also raise an exception, for example if the object received as their actual parameter is not of the expected class.
* equalsFunctionType  
  A behaviour type allowing the user of the collection to provide their implementation with an equality relation between key objects insofar that different object instances of the same content can be seen as equal which allows to ensure the uniqueness property for the keys as there can be no two different key instances k1, k2 where equalsFunction(k1.k2) is true.
* put  
  Adds a new keyElement - valueElement pair to the HashMap.  
  If the HashMap already contains a pair with the same keyELement, the old pair is removed before inserting the new pair.  
  Raises an exception in case of error, for example: running out of memory.
* get  
  Returns the valueElement part of a keyElement - valueElement pair in the HashMap, if such a pair with the provided keyElement object exists in the HashMap.  
  Raises an exception if the HashMap has no keyElement - valueElement pair with the provided keyElement.
* containsKey  
  Returns **true** if the HashMap contains a keyElement - valueElement pair with the provided keyElement, **false** otherwise.  
  Raises an exception in case of error, for example the hashFunction raised an exception.
* remove  
  Removes a keyElement - valueElement and returns the valueElement part of a keyElement - valueElement pair in the HashMap, if such a pair with the provided keyElement object exists in the HashMap.  
  Raises an exception in case of error, for example the hashFunction raised an exception.
* keyset  
  Returns a Set object containing a set of the keyElements of all the keyElement - valueElement pairs in the HashMap.
* values  
  Returns a List object containing the valueElement objects of all the keyElement - valueElement pairs in the HashMap.
* size  
  Returns the number of pairs stored in the HashMap.

## B.1.9 The Set class

The abstract [Set](#Set) class represents a set data structure for storing objects. This data structure is unordered and contains unique elements.  
A new Instance can be created via the external function **createSet**.

External function and class methods:

* createSet  
  Factory function for creating a new Set instance. It may be passed an equalsFunction to determine equality and ensure uniqueness of the contained set elements. Per default, instance equality is used.
* add  
  Adds an element to the Set if this is possible.  
  Returns true if the element could be added, returns false if the element was already present in the set and so was not added (to ensure uniqueness).  
  Raises an exception in case of error, for example: running out of memory.
* remove  
  Removes the provided element from the Set if it is present in the set.  
  Returns true if the element was located in the Set, false otherwise.  
  Subclasses might raise an exception.
* contains  
  Returns **true** if the Set contains the element, **false** otherwise.  
  Subclasses might raise an exception.
* iterator  
  Returns an iterator over the elements of this Set.  
  The elements are not iterated in any particular order.
* size  
  Returns the number of elements stored in the Set.

## B.1.10 The Exception class

The abstract [Exception](#Exception) class represents a generic exception that can be raised by standard collections.

Please note, that later the list of raised exception can be updated with more specific exceptions.

## B.1.11 The Iterator class

The abstract [Iterator](#Iterator) class represents an iterator over a collection. An instance of the Iterator class allows to iterate over the elements of a collection.

Class methods:

* hasNext  
  Returns true if the iterated collection still has elements not yet visited by the iterator.
* next  
  Returns the next element in the collection and steps the iterator for the upcoming collection.  
  Raises an exception if the collection has no more elements not yet visited.

# History

|  |  |  |
| --- | --- | --- |
| **Document history** | | |
| V1.1.1 | January 2019 | Publication |
| V1.2.1 | May 2020 | Publication |
| V1.3.1 | March 2021 | Membership Approval Procedure MV 20210515: 2021-03-16 to 2021-05-17 |
| V1.3.1 | May 2021 | Publication |
|  |  |  |