ETSI The Standards People

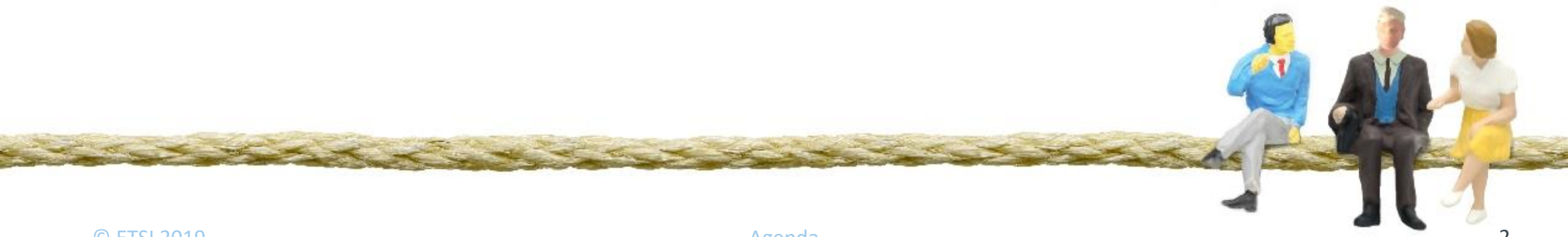
TTCN-3
Object-Oriented Features

Presented by: **STF573** For: **MTS#78**

10/11.09.2019

Agenda

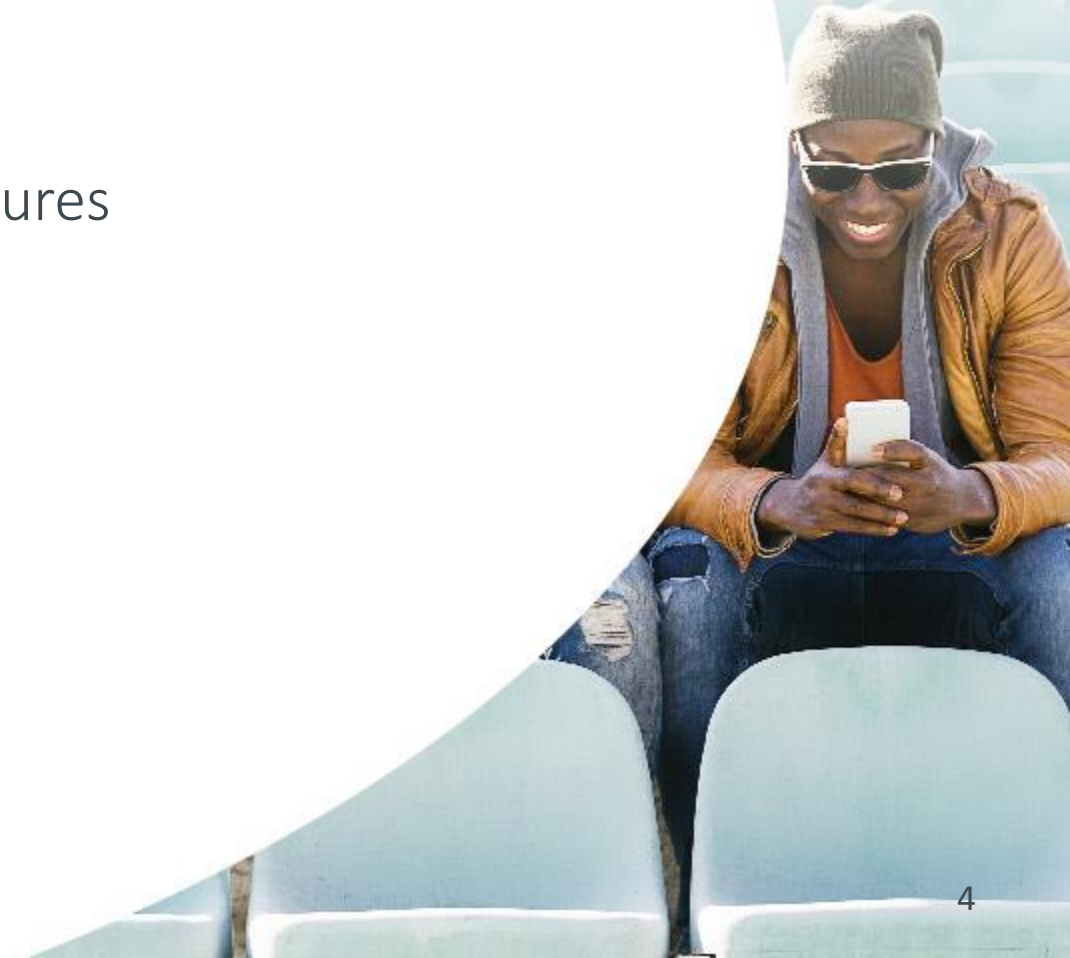
- Motivation and ideas
- Definition of class types using methods and fields
- Exception handling
- Real world examples



Motivation and ideas

Motivation and ideas

- Heighten appeal of TTCN-3 to users used to object-oriented programming
- Use advantages of object-oriented modelling
- Reduce TTCN-3 emulation of object-oriented features
- Allow simple access to external objects



Definition of class types

Definition of class types using methods and fields

- Inheritance and relationships between class definitions
- Visibility of class members
- Creation of class objects
- Discrimination of classes
- Casting

Simple class definition

- A *class* defines a new **TTCN-3 type**, containing one or more members:
  - Fields: `var`, `const`, `template`, `port`, `timer`
  - Methods: `function`, constructor (`create`)

```
type class MyDataClass {
    //fields
    var charstring v_dataType;

    //methods
    function f_classLog() {log(v_dataType);…};
    create() {this.v_dataType := "my initial string";}
    }
```

reference a member of own object (class)

Objects of classes (1)

```
type class MyDataClass {
    var charstring v_dataType;
    function f_classLog() {log(v_dataType);…};
    create() {this.v_dataType := "my initial string";}
    }
```

- An ***object*** is an instance (i.e. a *value*) of a *class*,
  - comprising a data *instance of each field* of the class,
  - created after invocation of the constructor of the class
  - can be created in a behavior running on a TTCN-3 component (the *owner* of the object)

Objects of classes (2)

```
type class MyDataClass {
    var charstring v_dataType;
    function f_classLog() {log(v_dataType);…};
    }
```

- Implicit constructor *(not needed to be specified)*:

```
create (charstring p_dataType) {this.v_dataType := p_dataType;}
```

- Creation of the object

```
var MyDataClass v_charObj := MyDataClass.create("1234");
v_charObj.f_classLog()
```

output is: 1234

Extension of classes (1)

```
type class MyDataClass {
    var charstring v_dataType;
    function f_classLog() {log(v_dataType);…};
    }

type class MyExtension extends MyDataClass {
    const charstring c_dataType2;
    }
```

superclass of MyExtension

subclass of MyDataClass

- A class definition inherits all declarations from its super class

Extension of classes (2)

```
type class MyDataClass {
    var charstring v_dataType;
    function f_classLog() {log(v_dataType);…};
    function @final f_classLog2() {log(v_dataType);…};
    }

type class @final MyExtension extends MyDataClass {
    const charstring c_dataType2;
    function f_classLog() {log(c_dataType2);…};}
```

overwriting!
No overloading

- *Final* classes can not have a subclass.
- Methods can be overwritten (if not declared as @final)

Extension of classes (3)

```
type class MyDataClass {
    var charstring v_dataType;
    }

type class MyExtension extends MyDataClass {
    var integer v_i;
    create (integer p_i):MyDataClass("1234") {this.v_i := p_i};
    public function f_classLog() {log(this.v_dataType);…};
    }
```

explicit own constructor

implicit super-constructor

- A constructor can refer to the (implicit) constructor of the superclass.

```
var MyExtension v_obj := MyExtension.create(55);
v_obj.f_classLog();
```

output is: 1234

Abstract classes

```
type class @abstract MyDataClass {
    var charstring v_dataType;
    function @abstract f_classLog();
    }

type class @final MyExtension extends MyDataClass {
    const charstring c_dataType2;
    function f_classLog() {log(v_dataType);…};
    }
```

implicit own constructor

implicit own constructor

- *Abstract* classes may include *abstract* member functions.
- NO explicit reference to constructor of *abstract* class: `MyDataClass.create`
- *Final* classes can not be abstract.

External classes

- *external* classes do only contain *external* functions.

```
external type class MyDataClass {
    function f_classLog();
    }

type class MyExtension extends MyDataClass {
    const charstring c_dataType2;
    function @external f_classLog2();
    }
```

keyword „*external*“ not needed

keyword „*external*“ needed

- Instantiation of an *object* of an external class implies the creation of an *external* object (*can* be shared between different parts of the test system: incl. racing conditions…!)
- The *internal* object of an external class has a reference to the *external* object (*handle*).

Visibility of members (1)

- Fields are *private* or *protected* (can not be overwritten) (default is protected)

```
type class MyDataClass {
    var charstring v_dataType;
    function f_log(charstring p_c) {log(v_dataType);…};
    }

type class MyExtension extends MyDataClass {
    var charstring v_dataType2;
    public function f_log(charstring p_c) {log(v_dataType & p_c);…};
    }
```

field is „protected“!

field is „protected“!

- *private* and *protected* members can not be accessed outside their class

```
var MyExtension v_charObj := MyExtension.create("1234");
v_charObj.f_log("56");
log(v_charObj.v_dataType);
```

output is: 123456

ERROR: no access to *private* field

Visibility of members (2)

- Methods are *private*, *protected* or *public* (default is protected)

```
type class MyDataClass {
    var charstring v_dataType := "1";
    function f_classLog(charstring p_c) {log(v_dataType);…};
    }

type class MyExtension extends MyDataClass {
    function f_classLog(charstring p_c) {log(v_dataType & p_c);…};
    public function f_log2() {super.f_classLog("2");};
    }
```

method is *„protected“!*

- *Public* member functions can only be overwritten by *public* member functions and can be called from any behavior on the object's owner component

```
var MyExtension v_charObj := MyExtension.create;
v_charObj.f_log2();
```

output is: 1

Visibility and usage of members

		subclasses (access)	subclasses (overwriting)	Object‘s owner component scope (access)
superclass member	**member visibility**			
fields	private	no	no	no
	protected	yes	no	no
constructor	protected	yes	no	no
	public	yes	yes	yes
function	private	no	no	no
	protected	yes	protected or public	no
	public	yes	public only	yes

Component type restrictions

- *runs on*, *system*, *mtc* restricts the component type that can create objects of that class (default is inherited from superclass) and shall be compatible with superclass clauses
- *function* members inherits restrictions from the (super)class (no own clauses)

```
type component MyComponent {
    port myport MyPortType;…
    }

type class MyClass runs on MyComponent system MySUT mtc MyTester {
    var charstring v_dataType;
    function f_classLog() {myport.receive;…};
    }
```

Class type discrimination

```
type class MyClass {…}
type class MyClassB extends MyClass {…}
```

- *of*-operator checks if most specific class of the ***object*** (*left-hand* side) is equal or subclass derived from the ***class type*** (*right-hand* side)
- *select class*-statement discriminates the class of an object (allows superclasses and subclasses of the object)

```
testcase TC_1() {…
    var MyClass v_a := MyClass.create;
    var MyClass v_b := MyClassB.create;
    if (v_a of MyClass) {…};
    select class (v_b) {
        case(MyClassB) {…}
        case(MyClassA) {…}
    }}
```

will be chosen

will not be chosen

Object reference

- To access an object *instance* an object *reference* is needed.
- An object reference is contained in a variable of a class type.
- The object is not copied when used as an actual parameter or assigned to a variable (only the reference). Multiple variables can contain a reference to the same object simultaneously.
- Objects cannot be shared by multiple components.

- Object references can be cast to another class

```
MyObject => NewClass;
MyObject => (NewClassReference);
```

Exception handling

Exception handling

- Exception type lists:
  functions, external functions, altsteps
- *raise* exception statements
- “catch” and “finally” clauses:
  statement blocks, altsteps and testcase

Extended behavior definition (overview)

	list of exception types	catch block	finally block
function	`exception (integer)`	`catch (integer p_e) { // do if exception // matches }`	`finally { // do before // termination }`
external function	`exception (integer, charstring, …)`		
altstep	`exception (MyExType)`	`catch (MyIntType p_e) { // first alternative } catch (integer p_e) { // second alternative }`	
testcase	n/a		
statement block			

raise Exception statement

- Causes **leaving** of: statement block, loop, alt, interleave
  - within the encompassing function/altstep/testcase

  (1) Execution **continues** in the *catch-block*
  - If encompassing function/altstep/testcase has *catch-block* (with same type, or can be cast)

  (2) Execution **leaves** function/altstep/testcase
  - If NO *catch block* available or can handle the raised exception
  - ➢ Handle the exception in the calling function/altstep/testcase

- Dynamic **error**, if exception not handled via catch clause of testcase statement block

Exception handling samples (1)

(1) execution continues in catch-block

```
function f_myf1() exception (integer) {…
    raise integer:1;
} catch (integer p_i) {…}
```

(2) execution continues outside

```
function f_myf1() exception (integer) {…
    raise integer:1;
}
```

Exception handling samples (2)

- execution continues in first catch-block that can handle the exception

```
function f_myf1() exception (integer, charstring) {…
   raise charstring: "exceptional state";
} catch (integer p_i) {…}
  catch (charstring p_c) {…
         raise integer:12;
        }
```

exception handled in *calling* function

- catch-block may raise another exception
  - to be handled outside the function

Final behavior

(1) execution continues in *catch*-block and *finally*-block

```
function f_myf1() exception (integer) {…
    raise integer:1;
} catch (integer p_i) {…}
  finally {…}
```

(2) execution continues in *finally*-block

```
function f_myf1() exception (integer) {…
    raise integer:1;
} finally {…}
```

exception handled outside function

Miscellaneous

Miscellaneous

- Module idenfication
- Built-in classes
- extended naming conventions (recommended)
  - *class definition:* Use upper-case initial letter
  - *object references:* **vo**_myObj

Module identification

- To indicate conformance to ETSI ES 203 790 V1.1.1

```
module myModule language "TTCN-3:2018 Object-Oriented" {…}
```

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:

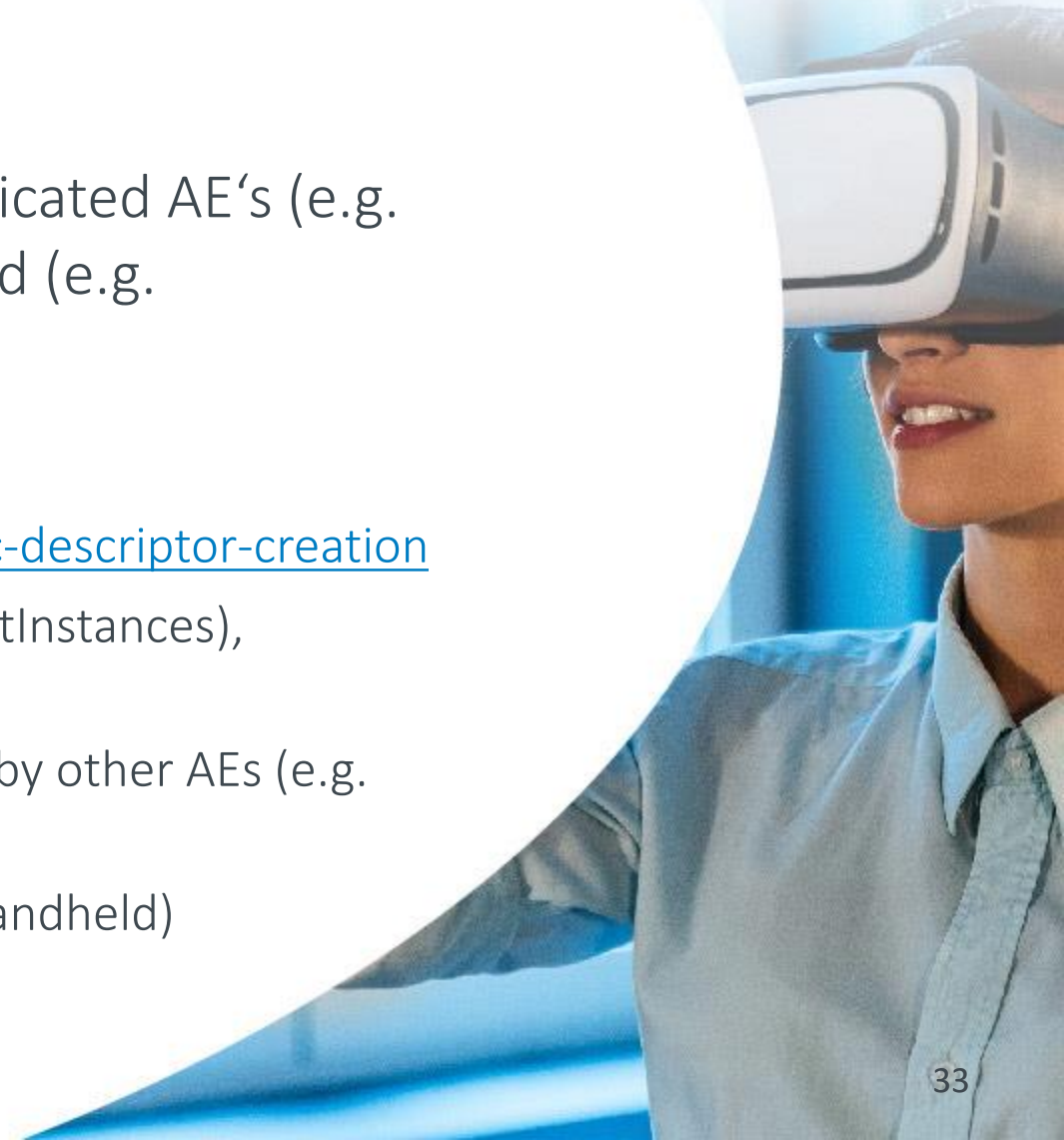
```
type class @abstract object {
   // This function will return a tool-specific descriptive string by default
   // but can be overridden by subclasses
   public function toString() return universal charstring;
   }
```

Real world examples

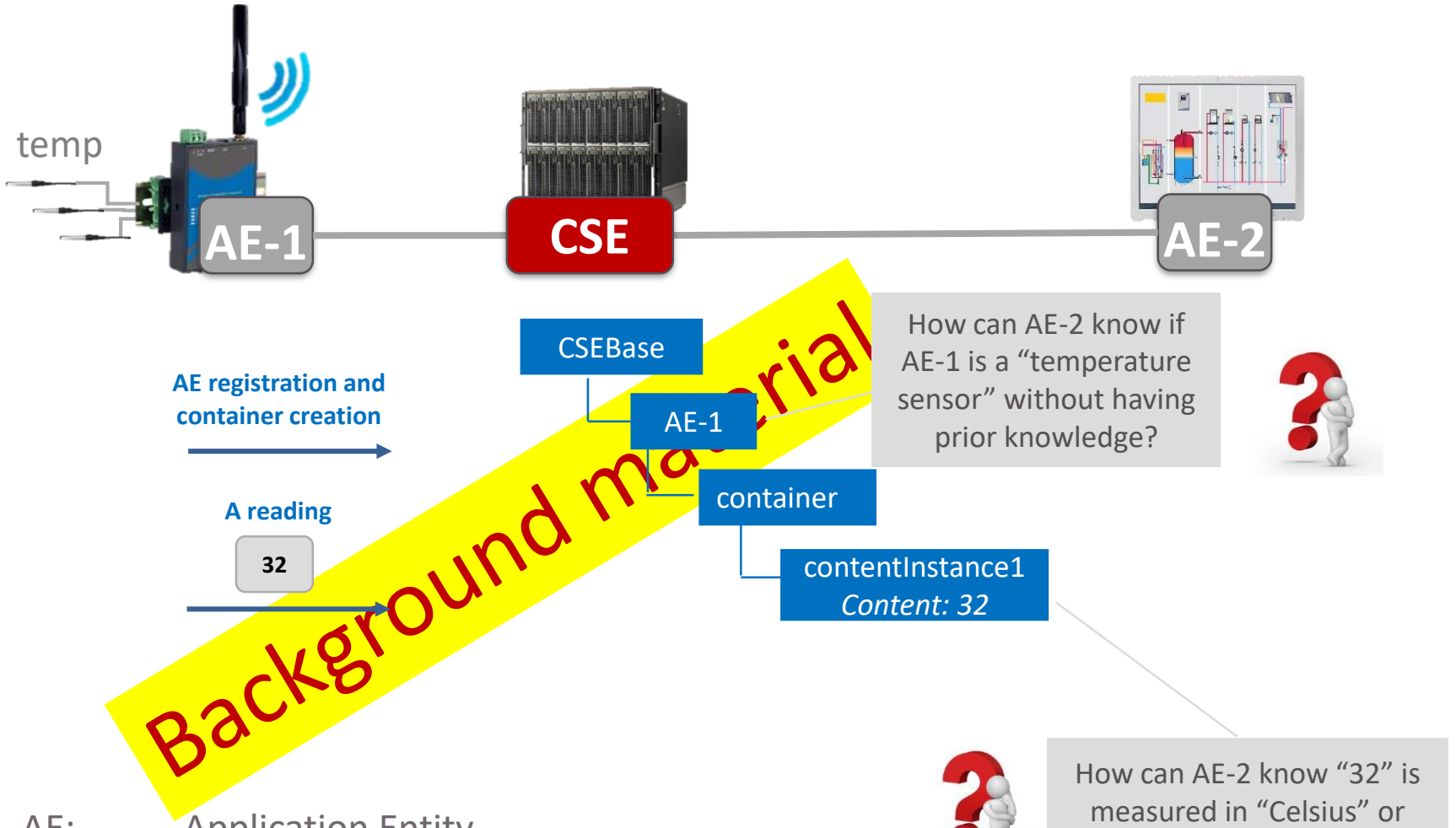
Real world examples

- oneM2M service/application elements
- E.g. semantics annotations/discovery: To find dedicated AE‘s (e.g. sensors), based on their location (e.g. area) or kind (e.g. temperature) etc.
- Possible scenario:
  - http://www.onem2m.org/tr-0045/procedures/semantic-descriptor-creation
  - creation of AE representations at CSE (container/contentInstances), e.g. temperature sensors
  - addition of semantic descriptors to AE representations, by other AEs (e.g. dashboard)
  - request semantic discovery, by other AEs (e.g. mobile handheld)

(work in progress)



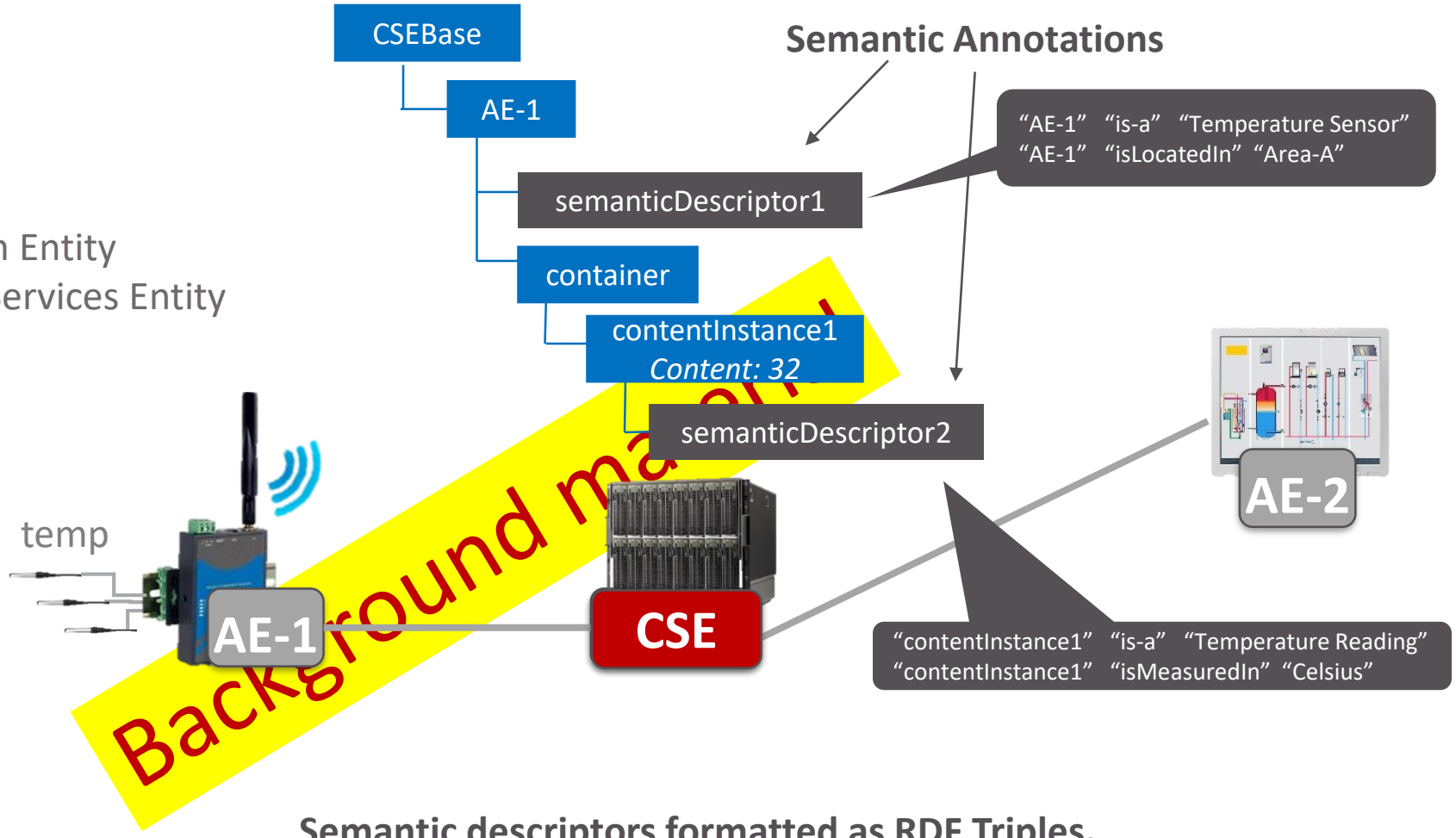
Semantic Annotation



AE: Application Entity
CSE: Common Services Entity

Semantic Annotation

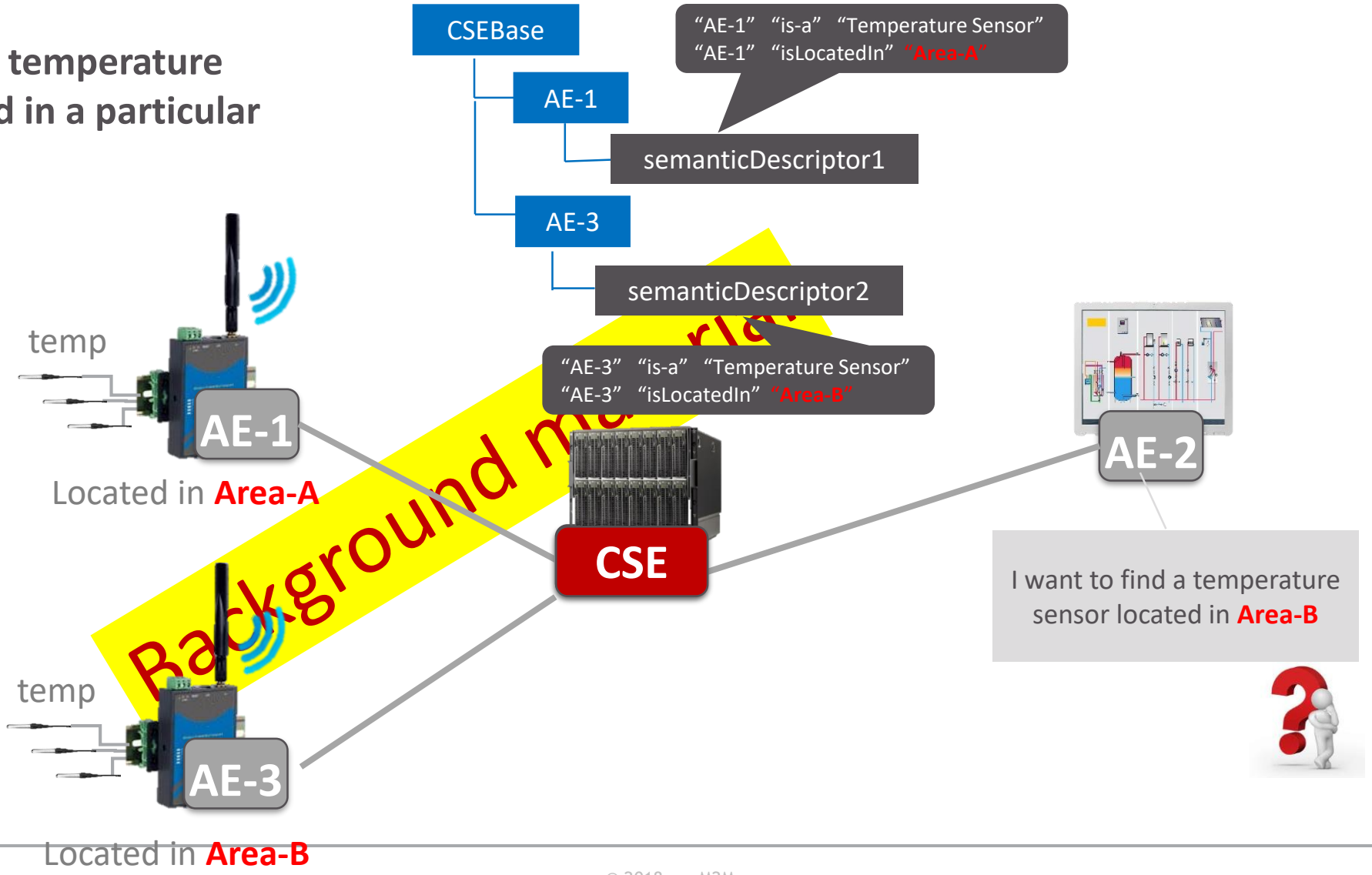
AE: Application Entity
CSE: Common Services Entity



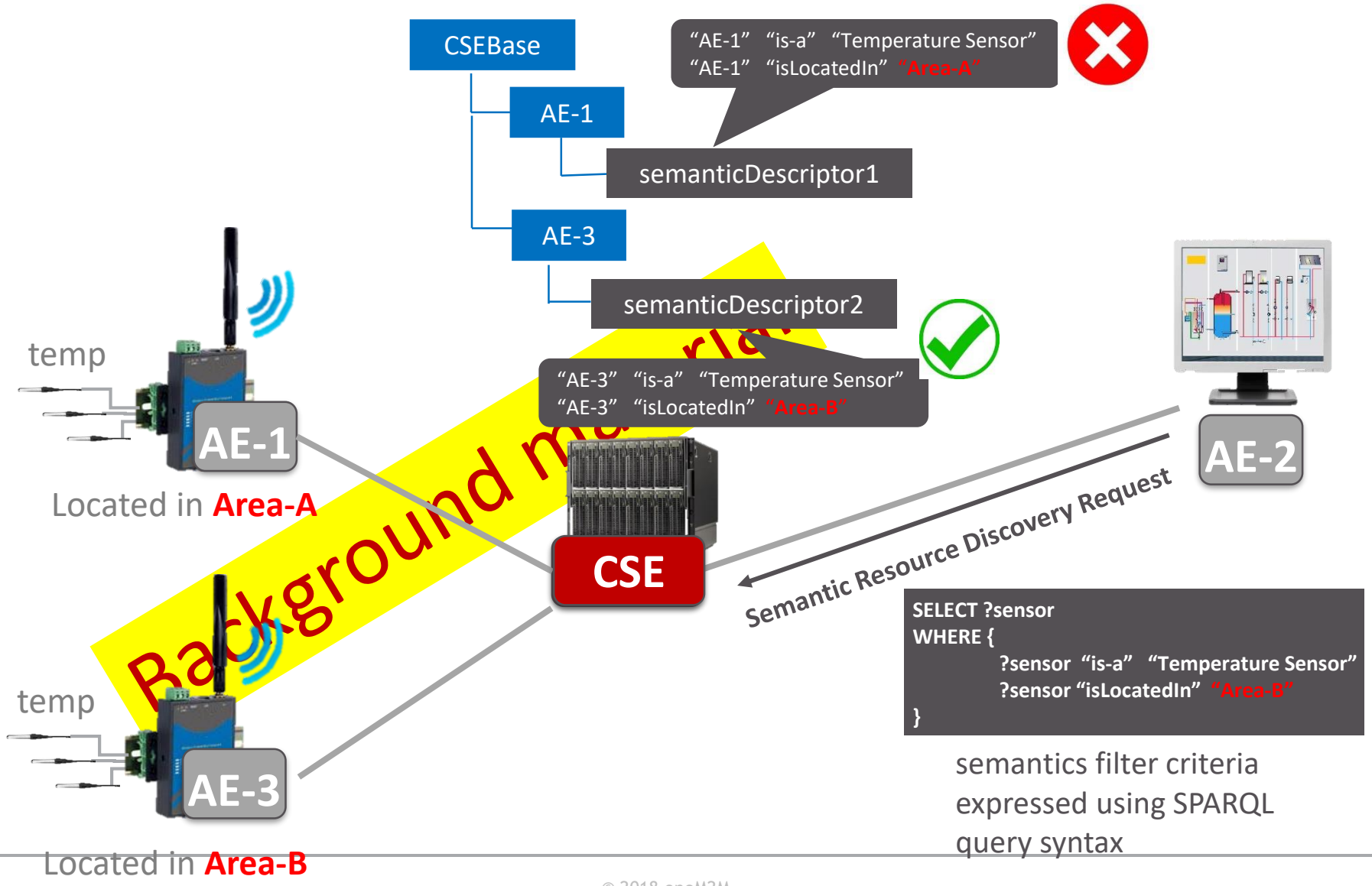
Semantic descriptors formatted as RDF Triples.
RDF triples expressed using ontology vocabularies (e.g. oneM2M Base Ontology, SAREF, etc)

Semantic Resource Discovery

Use Case – Find temperature sensor deployed in a particular location



Semantic Resource Discovery



SemanticDescriptor

```
type record SemanticDescriptor {
ResourceName resourceName,
ResourceType resourceType,
XSD.ID resourceID,
NhURI parentID,
Timestamp creationTime,
Timestamp lastModifiedTime,
Labels labels optional,
AcpType accessControlPolicyIDs optional,
Timestamp expirationTime,
ListOfURIs dynamicAuthorizationConsultationIDs optional,
ListOfURIs announceTo optional,
record length(1 .. infinity) of XSD.NCName announcedAttribute optio
XSD.ID creator optional,
SemanticFormat descriptorRepresentation optional,
Sparql semanticOpExec optional,
XSD.Base64Binary descriptor,
XSD.AnyURI ontologyRef optional,
ListOfURIs relatedSemantics optional,
XSD.Boolean semanticValidated opti
XSD.Boolean validationEnable o
union {
        record length(1 .. infinity)    urceRef childResource_list,
        record length(1 .. infinity) c    {
                Subscription subscription,
                Transaction transaction
        } choice_list
} choice optional
}
```