Karlsruhe Institute of Technology

Software Engineering Practice

WINTER TERM 2015/2016

rootJS

Node.js bindings for ROOT 6

Jonas Schwabe
Theo Beffart
Sachin Rajgopal
Christoph Wolff
Christoph Haas
Maximilian Früh

supervised by
Dr. Marek Szuba

Contents

1 CallbackHandler **3**
1.1 ctorCallback . . . 3
1.2 staticCtorCallback . . . 4
1.3 memberGetterCallback . . . 5
1.4 memberSetterCallback . . . 6
1.5 memberFunctionCallback . . . 7
1.6 staticGetterCallback . . . 8
1.7 staticSetterCallback . . . 9
1.8 staticFunctionCallback . . . 10

2 NodeHandler **11**
2.1 initialize . . . 11
2.2 getExports . . . 11

3 NodeApplication **12**
3.1 NodeApplication . . . 12

4 TemplateFactory **13**
4.1 createTemplate . . . 13

5 Proxy **15**
5.1 Proxy . . . 15
5.2 setAddress . . . 16
5.3 getAddress . . . 17
5.4 getType . . . 18
5.5 getScope . . . 19
5.6 isGlobal . . . 20
5.7 isTemplate . . . 21
5.8 isConst . . . 22
5.9 isStatic . . . 23

6 FunctionProxyFactory **24**
6.1 createFunctionProxy . . . 24
6.2 fromArgs . . . 25

7 FunctionProxy **26**
7.1 getCallFunc . . . 26
7.2 getMethodsFromName . . . 27
7.3 FunctionProxy . . . 28
7.4 getType . . . 29
7.5 validateArgs . . . 30
7.6 call . . . 31

8 ObjectProxyFactory **32**
8.1 createObjectProxy . . . 32

9 ObjectProxy **33**
9.1 ObjectProxy . . . 33
9.2 getType . . . 34
9.3 set . . . 35
9.4 get . . . 36
9.5 setProxy . . . 37
9.6 getProxy . . . 38

9.7 isPrimitive 39

10 Appendix **40**

10.1 Class diagram 40

10.2 Dynamic Model 42

10.3 Glossary 43

1. CallbackHandler

The CallbackHandler class gets invoked whenever an encapsulated ROOT function or object is accessed.

1.1. ctorCallback

Name	`CallbackHandler::ctorCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>* information about the context
Return value	**none**
Behavior	Gets invoked whenever a non static constructor function of an encapsulated ROOT class was called.

1.2. staticCtorCallback

Name	`CallbackHandler::staticCtorCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever a static constructor of an encapsulated ROOT class was called.

1.3. memberGetterCallback

Name	`CallbackHandler::memberGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, info: PropertyCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated (class) member was requested.

1.4. memberSetterCallback

Name	`CallbackHandler::memberSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated (class) member is attempted to be set.

1.5. memberFunctionCallback

Name	`CallbackHandler::memberFunctionCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo*$<$*Value*$>$
Return value	**none**
Behavior	Gets invoked whenever an non-static (class) function was called.

1.6. staticGetterCallback

Name	`CallbackHandler::staticGetterCallback(property: Local<String>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, info: PropertyCallbackInfo<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated static object was requested.

1.7. staticSetterCallback

Name	`CallbackHandler::staticSetterCallback(property: Local<String>, value: Local<Value>, info: PropertyCallbackInfo<Value>)`
Visibility	public
Parameters	*property: Local<String>, value: Local<Value>, info: PropertyCallback-Info<Value>*
Return value	**none**
Behavior	Gets invoked whenever an encapsulated static object is attempted to be set.

1.8. staticFunctionCallback

Name	`CallbackHandler::staticFunctionCallback(args: FunctionCallbackInfo<Value>)`
Visibility	public
Parameters	*args: FunctionCallbackInfo*$<$*Value*$>$
Return value	**none**
Behavior	Gets invoked whenever a static function was called.

2. NodeHandler

The *NodeHandler* is the main entry point when you require RootJS by using

```
// JavaScript: Load ROOT bindings in JavaScript
var root = require(rootJS.node);

// C++: Expose the initialize method as the main entry point
NODE_MODULE(rootJS, initialize)
```

after running the *initialize* method ROOT is fully initialized and all features are exposed to JavaScript.

2.1. initialize

Name	`NodeHandler::initialize(exports: Local<Object>, module: Local<Object>)`
Visibility	public static
Parameters	*exports: Local<Object>, module: Local<Object>* parameters passed by NodeJS
Return value	*none* The features will be exported by passing them to the exports parameter
Behavior	This will create an instance of *NodeApplication* and store it in gApplication, to ensure that all ROOT functionality that relies on gApplication will function properly. Further this will run *getExports* to retrieve the features to be exported to JavaScript which will then be put into the exports object which has been passed to this method

2.2. getExports

Name	`NodeHandler::getExports()`
Visibility	public
Parameters	*none*
Return value	**Local<Object>** features to be exported
Behavior	This method will run multiple private methods to collect global functions, global variables, macros and classes. All these items will be stored in a v8 object which will be passed to RootJS via the initialize method.

3. NodeApplication

ROOT uses TApplication to interface with the windowing system and event handlers. An insatnce of TApplication is usually stored in the global *gApplication* variable.

The main problem with using TApplication directly would be, that we could not hook into the *InitializeGraphics* method. When having a graphical user interface we need to do a UI update frequently:

```
gSystem->ProcessEvents();
```

To avoid having a lot of *ProcessEvents* calls, we wait until *InitializeGraphics* has been called at least once.

Further NodeApplication is being used to set the application's name and initialize a custom message callback which can be used to retrive messages in JavaScript.

3.1. NodeApplication

Name	`NodeApplication::NodeApplication(acn: char*, argc: int*, argv: char**)`
Visibility	public
Parameters	*acn: char*, argc: int*, argv: char***
Return value	**«constructor»** describe return value
Behavior	Set's the application name and constructs a custom message handler

4. TemplateFactory

Creates Javascript function templates from a given ROOT class using *TClassRef*. Methods and static members are set during creation through the use of ROOT reflections and the proxy factories. The created templates are kept in a cache to avoid unnecessary creation of already existing templates.

4.1. createTemplate

Name	`TemplateFactory::createTemplate(clazz: TClassRef)`
Visibility	public
Parameters	*clazz: TClassRef* the class for which a template is to be created
Return value	**Local<FunctionTemplate>** the created template
Behavior	gets the class from TClassRef and creates a new function template. then it iterates over all static members of the class and sets the corresponding members of the template to respective proxy objects. It then iterates through the functions and also sets them. For further reference consider the following sequence diagram.

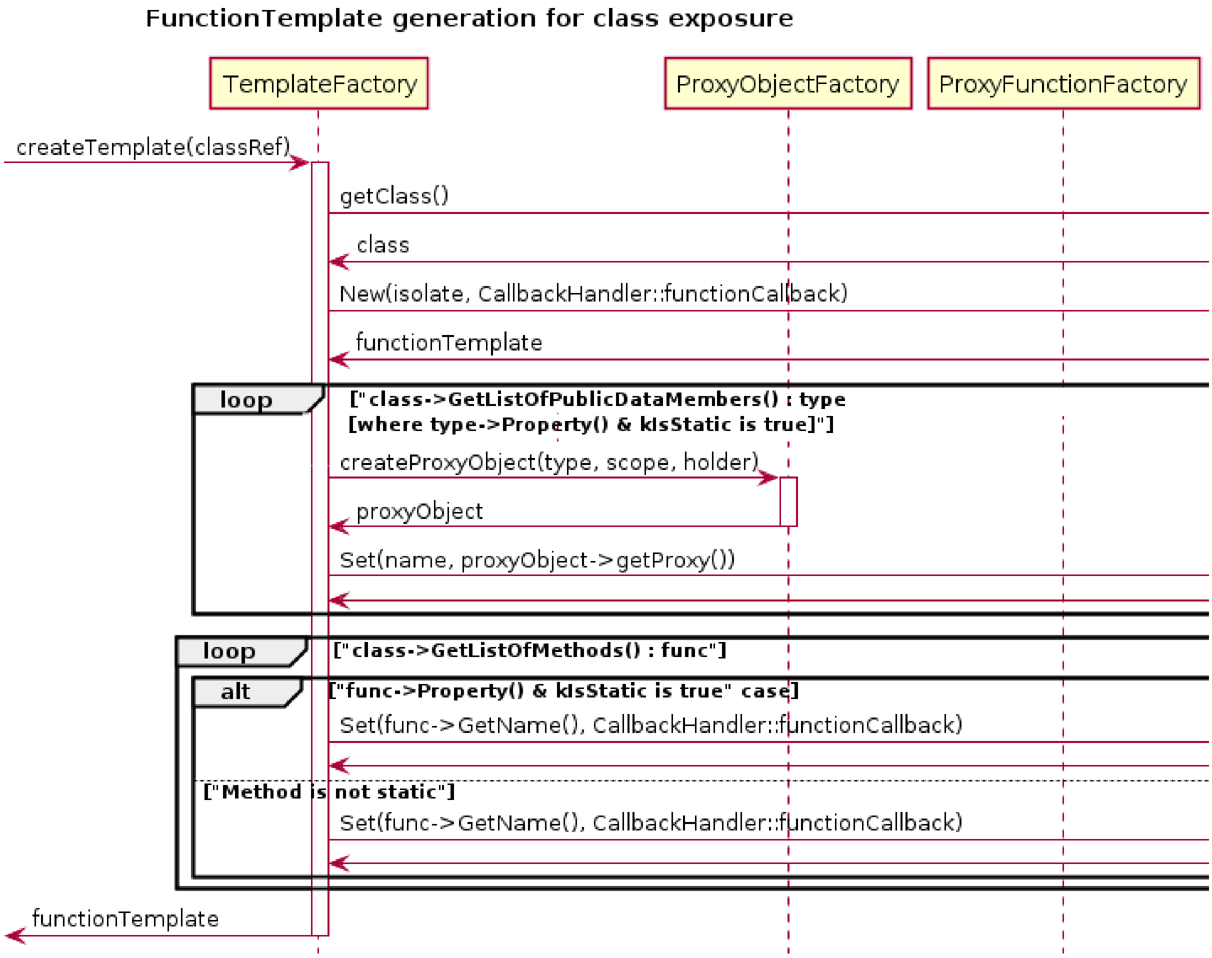

Figure 4.1: function template creation (full diagram in appendix)

5. Proxy

The *Proxy class* is an abstract class which acts as an intermediary between Node.js and ROOT. Both the *ObjectProxy* and *FunctionProxy* inherit the *Proxy* class, since both of them require the object's or function's *void** address, *TObject* type and *TClassRef* scope. The *Proxy* class holds the data, which both *ObjectProxy* and *FunctionProxy* require. The *Proxy* class uses the Proxy design pattern.

5.1. Proxy

Name	`Proxy::Proxy(address: void*, type: TObject, scope: TClassRef)`
Visibility	protected
Parameters	*address: void** The memory address of the Proxy *type: TObject* The type identification which the ObjectProxy will have *scope: TClassRef* The class the Proxy belongs to
Return value	**«constructor»** Returns a Proxy with the given parameters as a variables
Behavior	The Proxy constructor will be inherited by both ObjectProxy and FunctionProxy. The created Proxy will have the parameters as variables.

5.2. setAddress

Name	`Proxy::setAddress(address: void*)`
Visibility	public
Parameters	*address: void** The address to which the Proxy should be set to
Return value	**none**
Behavior	Sets the address of the Proxy.

5.3. getAddress

Name	`Proxy::getAddress()`
Visibility	public
Parameters	*none*
Return value	**void*** The current address of the Proxy
Behavior	Gets the current address of the Proxy.

5.4. getType

Name	`Proxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TObject** The current type of the Proxy
Behavior	Gets the current type of the Proxy.

5.5. getScope

Name	`Proxy::getScope()`
Visibility	public
Parameters	*none*
Return value	**TClassRef** The current scope of the Proxy
Behavior	Gets the current scope of the Proxy.

5.6. isGlobal

Name	`Proxy::isGlobal()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the Proxy is global
Behavior	Checks if the Proxy is global and hence visible throughout the program.

5.7. isTemplate

Name	`Proxy::isTemplate()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the Proxy is a template
Behavior	Checks if the Proxy is a template, which allows using generic types.

5.8. isConst

Name	`Proxy::isConst()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the Proxy is a constant
Behavior	Checks if the Proxy is a constant.

5.9. isStatic

Name	`Proxy::isStatic()`
Visibility	public
Parameters	*none*
Return value	**bool** True if the Proxy is static
Behavior	Checks if the Proxy is static.

6. FunctionProxyFactory

The *FunctionProxyFactory*

6.1. createFunctionProxy

Name	`FunctionProxyFactory::createFunctionProxy(function: TFunction, scope: TClassRef)`
Visibility	public
Parameters	*function: TFunction* The ROOT function to be proxied. *scope: TClassRef*
Return value	**ProxyFunction** the proxied function
Behavior	Creates *FunctionProxy* objects with *TFunction* function and *TClassRef* scope.

6.2. fromArgs

Name	`FunctionProxyFactory::fromArgs(name: string, scope: TClassRef, args: FunctionCallbackInfo)`
Visibility	public
Parameters	*name: string, scope: TClassRef, args: FunctionCallbackInfo* The name of the function, its scope and the arguments it takes.
Return value	**FunctionProxy** describe return value
Behavior	This method is called from the *createFunctionProxy* method to deal with overloaded functions, since JavaScript doesn't support it.

7. FunctionProxy

Acts as a proxy for a ROOT callable (i.e. function or class method). It provides methods to execute such a callable and validate its arguments. It also maintains a map of *TFunction* - *CallFunc* entries to cache already used functions.

7.1. getCallFunc

Name	`FunctionProxy::getCallFunc(method: TFunction*)`
Visibility	public
Parameters	*method: TFunction**: pointer to the ROOT function for which a proxy is to be created
Return value	**CallFunc*** a pointer to the CallFunc object provied by kling
Behavior	gets a pointer to a CallFunc object, which encapsulates the provided TFunction in storage (CallFunc is made available by cling) to which is used during this class' instantiation

7.2. getMethodsFromName

Name	`FunctionProxy::getMethodsFromName(scope: TClassRef, name: string)`
Visibility	public
Parameters	*scope: TClassRef* a reference to the class which is checked for methods with the specified name *name: string* the name of the overloaded methods which shall be returned
Return value	**vector<TFunction*>** all methods that match the specified name
Behavior	Gets a reference to a class and a method name string. It returns all methods of the class with the specified name. This is needed since JavaScript does not support method overloading.

7.3. FunctionProxy

Name	`FunctionProxy::FunctionProxy(address: void*, function: TFunction, scope: TClassRef)`
Visibility	public
Parameters	*address: void** the memory address of the proxied function *function: TFunction* the function's reflection object *scope: TClassRef* the class that the function belogs to
Return value	**«constructor»** the created `FunctionProxy`
Behavior	Creates the `FunctionProxy`.

7.4. getType

Name	`FunctionProxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TFunction** the TFunction object in the proxy
Behavior	returns the TFunction object this proxy wraps. It contains the meta data of its corresponding function

7.5. validateArgs

Name	`FunctionProxy::validateArgs(args: FunctionCallbackInfo)`
Visibility	public
Parameters	*args: FunctionCallbackInfo* information about the context of the call, including the number and values of arguments
Return value	**ObjectProxy[]** array of the arguments as proxies
Behavior	checks whether the function is being called with the proper arguments and wraps them in proxies so they can be used by the call method

7.6. call

Name	`FunctionProxy::call(args: ObjectProxy[])`
Visibility	public
Parameters	*args: ObjectProxy[]* proxies containing arguments for the method
Return value	**ObjectProxy** proxies containing the values returned by the called method
Behavior	calls the actual method in storage using cling. The argument object proxies' contents are read and given to the called method

8. ObjectProxyFactory

The *ObjectProxyFactory* creates *ObjectProxy* instances with *TDataMember* type, *TClassRef* scope and *ObjectProxy* holder. It encapsulates ROOT objects recursively for use in Javascript.

8.1. createObjectProxy

Name	`ObjectProxyFactory::createObjectProxy(type: TDataMember, scope: TClassRef, holder: ObjectProxy)`
Visibility	public
Parameters	*type: TDataMember* The type identification which the ObjectProxy will have *scope: TClassRef* The class the ObjectProxy belongs to *holder: ObjectProxy* The holder is the ObjectProxy which will encapsulate and hold the newly created ObjectProxy
Return value	**ObjectProxy** Returns the ObjectProxy which is created with the given parameters. 1q
Behavior	A new ObjectProxy is created each time the createObjectProxy method is called up.

9. ObjectProxy

The *ObjectProxy* class is used to represent ROOT objects. It differentiates between primitive and non-primitive object types.
There are the following implementations of *ObjectProxy*:

- **EnumProxy** Maps C++ enums to JavaScript strings
- **StructProxy** Maps C++ structs to JavaScript objects
- **ArrayProxy** Maps C++ arrays to JavaScript arrays, we cannot enlarge C++ arrays, so we will throw an Exception on overflows
- **PointerProxy** Maps C++ pointers to JavaScript objects
- **NumberProxy** Uses a C++ template to map all C++ numbers to JavaScript Numbers
- **StringProxy** Maps C++ strings and c-strings to JavaScript strings
- **BooleanProxy** Maps C++ root boolean to Javascript boolean

The *ObjectProxyFactory* decides which *ObjectProxy* needs to be instantiated. Internally all these *ObjectProxies* work the same way by linking a *v8::Local* with a *TDataMember*

9.1. ObjectProxy

Name	`ObjectProxy::ObjectProxy(type: TDataMember, scope: TClassRef)`
Visibility	public
Parameters	*type: TDataMember, scope: TClassRef* the type and scope of the object
Return value	**«constructor»** the newly constructed ObjectProxy
Behavior	Creates a new ObjectProxy with the given type and scope.

9.2. getType

Name	`ObjectProxy::getType()`
Visibility	public
Parameters	*none*
Return value	**TDataMember** the type of the ObjectProxy
Behavior	Returns the type of the Object behind the proxy.

9.3. set

Name	`ObjectProxy::set(value: ObjectProxy)`
Visibility	public
Parameters	*value: ObjectProxy* the value to set
Return value	**none**
Behavior	Sets the value of the Object behind the proxy.

9.4. get

Name	`ObjectProxy::get()`
Visibility	public
Parameters	*none*
Return value	**Local**<**Value**> The value the object has.
Behavior	Returns the value that was set for the object.

9.5. setProxy

Name	`ObjectProxy::setProxy(proxy: Local<Object>)`
Visibility	public
Parameters	*proxy: Local<Object>*
Return value	**none**
Behavior	describe beahviour

9.6. getProxy

Name	`ObjectProxy::getProxy()`
Visibility	public
Parameters	*none*
Return value	**Local**<**Object**> describe return value
Behavior	describe beahviour

9.7. isPrimitive

Name	`ObjectProxy::isPrimitive()`
Visibility	public
Parameters	*none*
Return value	**bool** Whether or not the represented object is of a primitive type or not.
Behavior	Returns *true* if the represented object's type is primitive, *false* if not.

10. Appendix

10.1. Class diagram

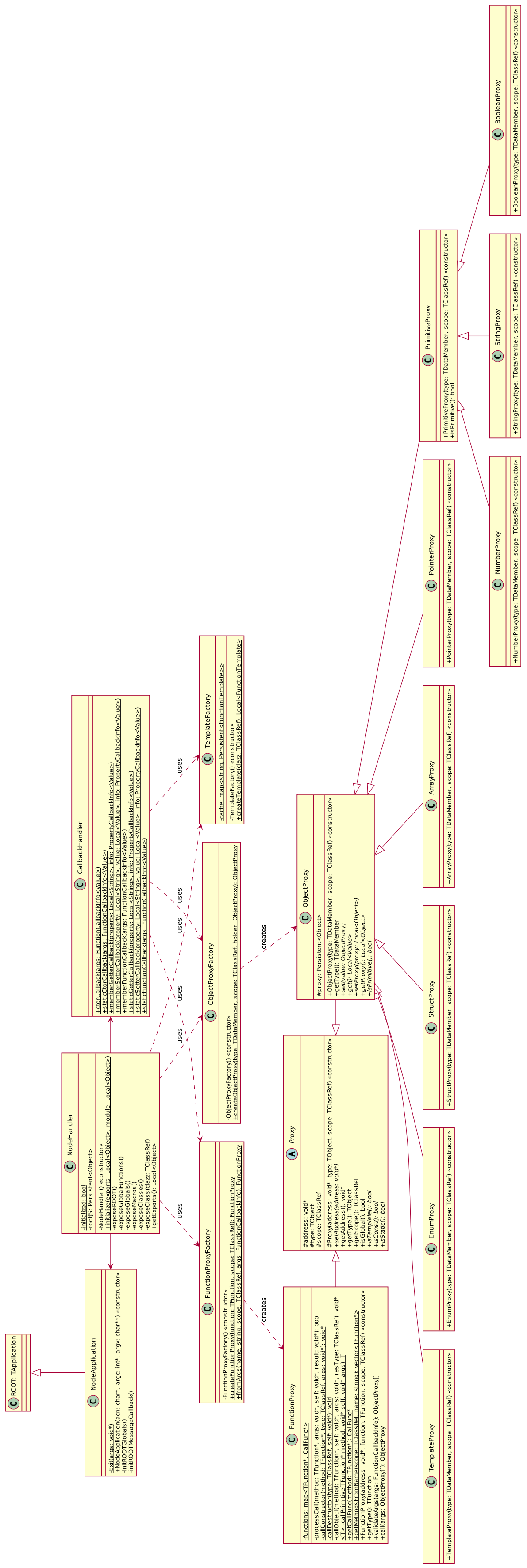

Figure 10.1: rootJS class diagram

10.2. Dynamic Model

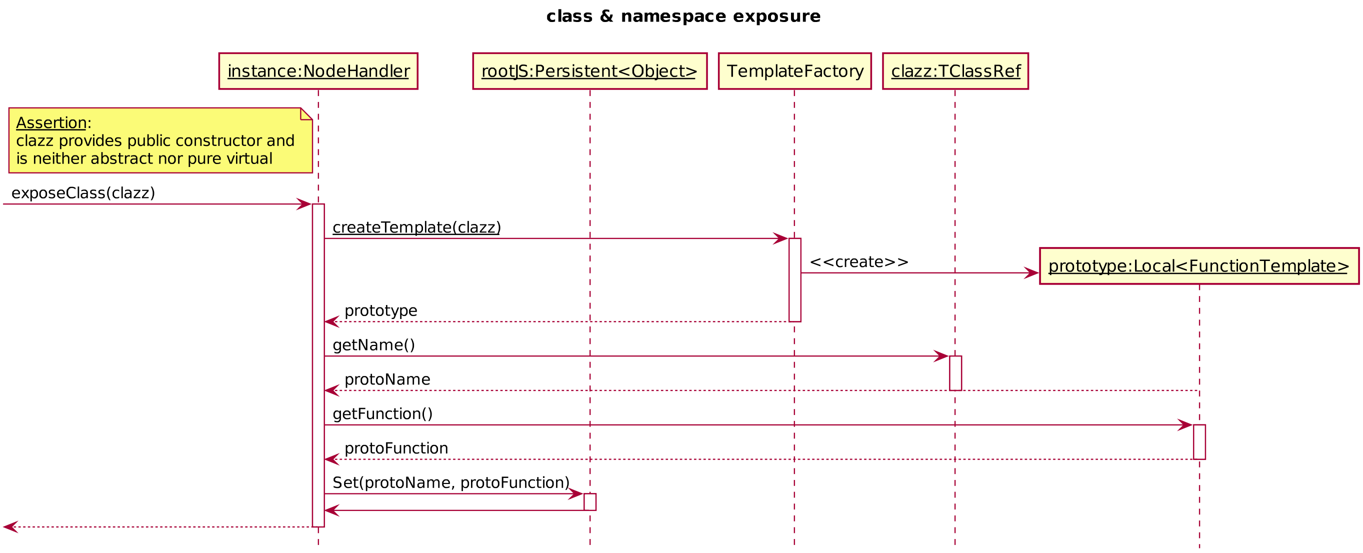

Figure 10.2: class exposure sequence

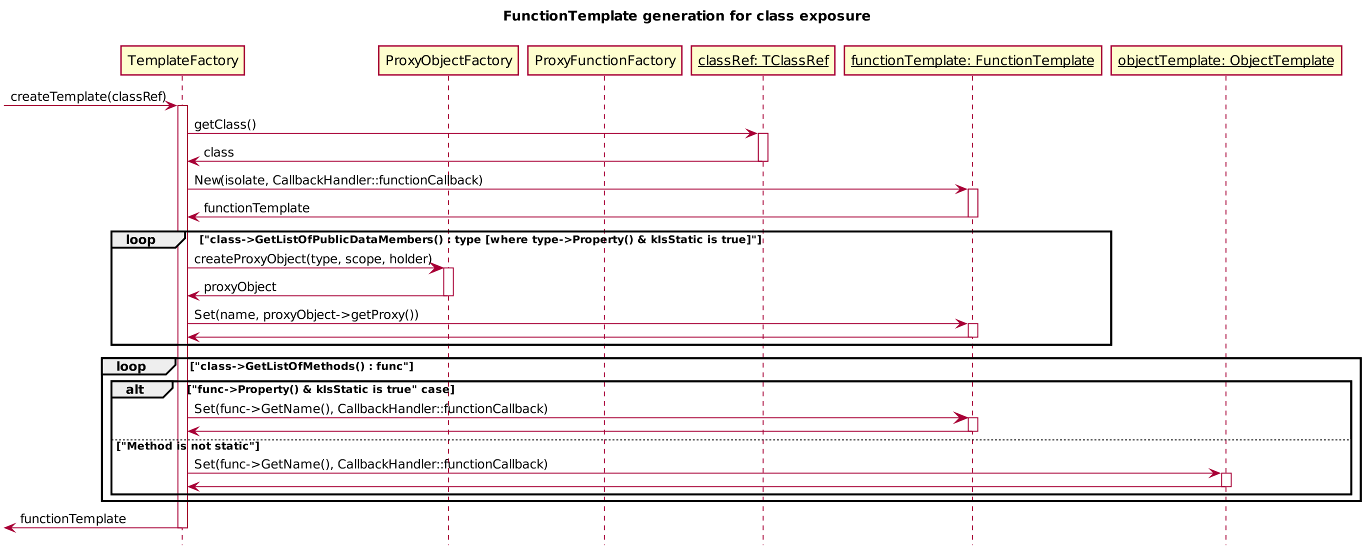

Figure 10.3: class exposure sequence

10.3. Glossary