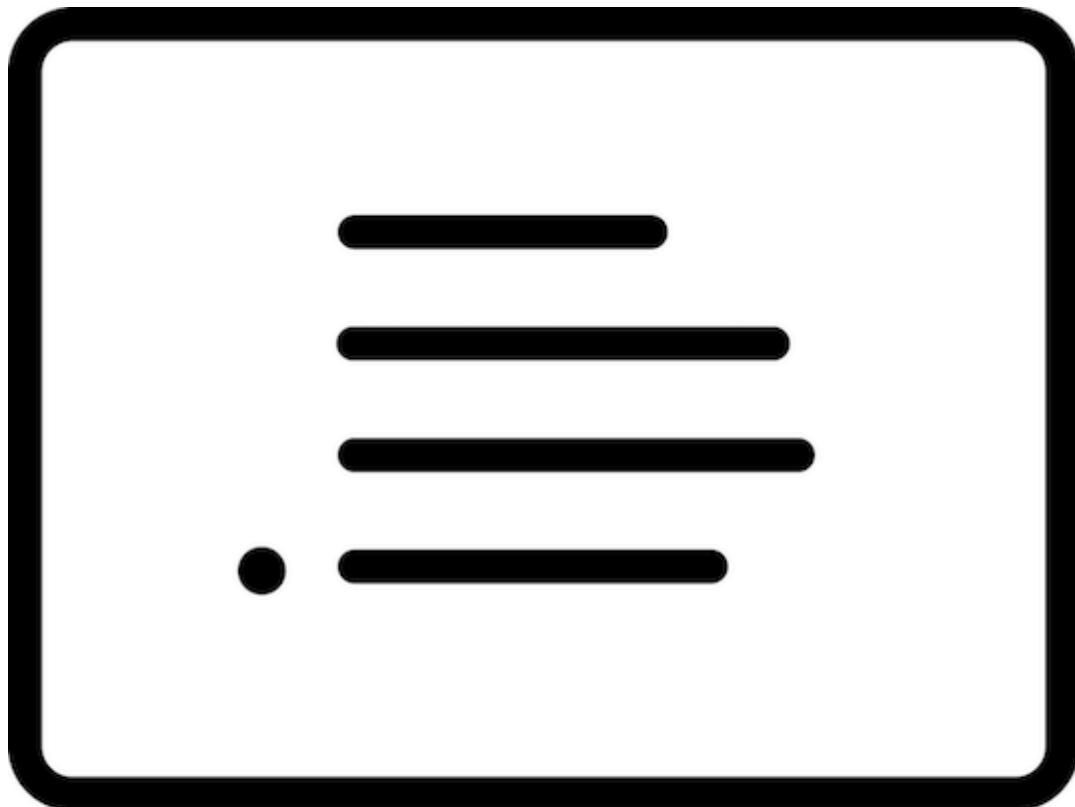


# N4JS Language Specification



# N4JS

2018-11-06 16:56:07 CET



# Table of Contents

	xvii
1. Introduction .....	1
1.1. Notation .....	1
1.1.1. Grammar Notation .....	1
1.1.2. Type Judgments and Rules and Constraints Notation .....	1
1.1.2.1. Typing Rules and Judgments .....	1
1.1.2.2. Types of an Element .....	2
1.2. Auxiliary Functions .....	2
1.2.1. Binding .....	2
1.2.2. Merging Types .....	3
1.2.2.1. Logic Formulas .....	3
1.2.2.3. Symbols and Font Convention .....	3
2. Grammar .....	5
2.1. Lexical Conventions .....	5
2.1.1. Identifier Names and Identifiers .....	5
2.1.2. This Keyword .....	6
2.1.3. Regular Expression Literals .....	6
2.1.4. Automatic Semicolon Insertion .....	6
2.1.5. JSDoc .....	6
3. Names .....	7
3.1. Access Control .....	7
3.2. Accessibility of Types, Top-Level Variables and Function Declarations .....	7
3.2.1. Accessibility of Members .....	9
3.2.2. Valid Names .....	12
3.2.3. Qualified Names .....	13
3.2.4. Name Duplicates .....	13
3.2.4.1. Lexical Environment .....	13
3.2.4.2. Duplicates and Shadowing .....	14
4. Types .....	17
4.1. Overview .....	17
4.2. Type Expressions .....	19
4.2.1. Syntax .....	19
4.2.2. Properties .....	19
4.2.3. Semantics .....	20
4.3. Type Inference .....	20
4.4. Generic and Parameterized Types .....	21
4.4.1. Generic Types .....	21
4.4.2. Type Variables .....	21
4.4.3. Parameterized Types .....	23
4.5. Primitive ECMAScript Types .....	27
4.5.1. Undefined Type .....	28
4.5.2. Null Type .....	28
4.5.3. Primitive Boolean Type .....	29
4.5.4. Primitive String Type .....	29
4.5.5. Primitive Number Type .....	29
4.5.6. Primitive Type int .....	30
4.5.7. Primitive Symbol Type .....	30
4.6. Primitive N4JS Types .....	30
4.6.1. Any Type .....	30
4.6.1.1. Semantics .....	30
4.6.1.2. Type Inference .....	30
4.6.2. Void Type .....	31
4.6.2.1. Semantics .....	31
4.6.3. Unknown Type .....	31
4.6.4. Primitive Pathselector and I18nKey .....	31
4.6.4.1. Semantics .....	32
4.7. Built-in ECMAScript Object Types .....	32

4.7.1. Semantics .....	32
4.7.2. Object Type .....	32
4.7.3. Function-Object-Type .....	33
4.7.4. Array Object Type .....	33
4.7.5. String Object Type .....	33
4.7.6. Boolean Object Type .....	33
4.7.7. Number Object Type .....	33
4.7.8. Global Object Type .....	33
4.7.9. Symbol .....	33
4.7.10. Promise .....	33
4.7.11. Iterator Interface .....	34
4.7.12. Iterable Interface .....	34
4.8. Built-In N4JS Types .....	34
4.8.1. N4Object .....	34
4.8.1.1. Semantics .....	34
4.8.2. N4Class .....	34
4.8.3. IterableN .....	34
4.9. Type Modifiers .....	34
4.9.1. Dynamic .....	35
4.9.2. Optional Return Types .....	35
4.10. Union and Intersection Type (Composed Types) .....	35
4.10.1. Union Type .....	36
4.10.1.1. Syntax .....	36
4.10.1.2. Semantics .....	36
4.10.1.3. Warnings .....	39
4.10.2. Intersection Type .....	39
4.10.2.1. Syntax .....	39
4.10.2.2. Semantics .....	39
4.10.2.3. Warnings .....	41
4.10.3. Composed Types in Wildcards .....	41
4.10.4. Property Access for Composed Types .....	42
4.10.4.1. Properties of Union Type .....	42
4.10.4.2. Properties of Intersection Type .....	43
4.11. Constructor and Classifier Type .....	45
4.11.1. Syntax .....	45
4.11.2. Semantics .....	45
4.11.3. Constructors and Prototypes in ECMAScript 2015 .....	47
4.12. This Type .....	48
4.12.1. Syntax .....	49
4.12.2. Semantics .....	49
4.13. Enums .....	51
4.13.1. Enums (N4JS) .....	51
4.13.1.1. Syntax .....	51
4.13.1.2. Semantics .....	52
4.13.2. String-Based Enums .....	53
5. Classifiers .....	57
5.1. N4JS Specific Classifiers .....	57
5.1.1. Properties .....	57
5.1.2. Common Semantics of Classifiers .....	57
5.1.3. Classes .....	58
5.1.3.1. Definition of Classes .....	58
5.1.3.2. Semantics .....	59
5.1.3.3. Final Modifier .....	60
5.1.3.4. Abstract Classes .....	61
5.1.3.5. Non-Instantiable Classes .....	61
5.1.3.6. Superclass .....	61
5.1.4. Interfaces .....	62
5.1.4.1. Definition of Interfaces .....	62
5.1.5. Generic Classifiers .....	64
5.1.6. Definition-Site Variance .....	65

5.2. Members .....	67
5.2.1. Syntax .....	67
5.2.1.1. Properties .....	67
5.2.2. Semantics .....	68
5.2.3. Methods .....	68
5.2.3.1. Syntax .....	68
5.2.3.2. Properties .....	69
5.2.3.3. Semantics .....	70
5.2.3.4. Final Methods .....	71
5.2.3.5. Abstract Methods .....	72
5.2.3.6. Generic Methods .....	72
5.2.4. Default Methods in Interfaces .....	72
5.2.4.1. Asynchronous Methods .....	73
5.2.5. Constructors .....	73
5.2.5.1. Structural This Type in Constructor .....	74
5.2.5.2. @Spec Constructor .....	75
5.2.5.3. Callable Constructors .....	79
5.2.5.4. Covariant Constructors .....	79
5.2.6. Data Fields .....	80
5.2.6.1. Syntax .....	80
5.2.6.2. Properties .....	81
5.2.6.3. Assignment Modifiers .....	82
5.2.6.4. Field Accessors (Getter/Setter) .....	83
5.2.6.5. Optional Fields .....	84
5.2.7. Static Members .....	87
5.2.7.1. Access From and To Static Members .....	88
5.2.7.2. Generic static methods .....	89
5.2.7.3. Static Members of Interfaces .....	89
5.2.8. Redefinition of Members .....	90
5.2.8.1. Overriding of Members .....	91
5.2.8.2. Implementation of Members .....	91
5.3. Structural Typing .....	94
5.3.1. Syntax .....	95
5.3.2. Definition Site Structural Typing .....	95
5.3.3. Use-Site Structural Typing .....	96
5.3.4. Structural Read-only, Write-only and Initializer Field Typing .....	101
5.3.5. Public Setter Annotated With <code>ProvidesInitializer</code> .....	102
5.3.6. Structural Types With Optional Fields .....	102
5.3.7. Structural Types With Access Modifier .....	102
5.3.8. Structural Types With Additional Members .....	103
5.3.8.1. Syntax .....	103
6. Functions .....	105
6.1. Function Type .....	105
6.1.1. Properties .....	105
6.1.2. Type Inference .....	106
6.1.3. Autoboxing of Function Type .....	109
6.1.4. Arguments Object .....	110
6.2. ECMAScript 5 Function Definition .....	111
6.2.1. Function Declaration .....	111
6.2.1.1. Syntax .....	111
6.2.1.2. Semantics .....	112
6.2.2. Function Expression .....	112
6.2.2.1. Syntax .....	112
6.2.2.2. Semantics and Type Inference .....	112
6.3. ECMAScript 2015 Function Definition .....	113
6.3.1. Formal Parameters .....	113
6.3.1.1. Optional Parameters .....	113
6.3.1.2. Default Parameters .....	113
6.3.1.3. Variadic .....	113
6.3.2. Generator Functions .....	114

6.3.2.1. Syntax .....	114
6.3.2.2. Semantics .....	114
6.3.2.3. Generator Arrow Functions .....	116
6.3.3. Arrow Function Expression .....	116
6.3.3.1. Syntax .....	116
6.3.3.2. Semantics and Type Inference .....	116
6.4. ECMAScript Proposals Function Definition .....	117
6.4.1. Asynchronous Functions .....	117
6.4.1.1. Syntax .....	117
6.4.1.2. Semantics .....	118
6.4.1.3. Asynchronous Arrow Functions .....	119
6.5. N4JS Extended Function Definition .....	119
6.5.1. Generic Functions .....	119
6.5.2. Promisifiable Functions .....	119
7. Conversions and Reflection .....	123
7.1. Autoboxing and Coercing .....	123
7.1.1. Coercing .....	123
7.1.2. Autoboxing of Primitives .....	123
7.1.3. Autoboxing of Function Expressions and Declarations .....	124
7.2. Auto-Conversion of Objects .....	124
7.2.1. Auto-Conversion of Class Instances .....	124
7.2.1.1. Auto-Conversion of Interface Instances .....	125
7.2.2. Auto-Conversion of Enum Literals .....	125
7.3. Type Cast and Type Check .....	125
7.3.1. Type Cast .....	125
7.3.2. Type Check .....	126
7.4. Reflection meta-information .....	126
7.4.1. Reflection for Classes .....	127
7.4.2. Reflection for Interfaces .....	128
7.4.3. Reflection for Enumerations .....	128
7.5. Conversion of primitive types .....	128
8. Expressions .....	129
8.1. ECMAScript 5 Expressions .....	129
8.1.1. The this Literal .....	129
8.1.2. Identifier .....	133
8.1.3. Literals .....	134
8.1.3.1. Integer Literals .....	134
8.1.4. Array Literal .....	135
8.1.5. Object Literal .....	136
8.1.5.1. Properties .....	137
8.1.5.2. Scoping and linking .....	138
8.1.6. Parenthesized Expression and Grouping Operator .....	139
8.1.7. Property Accessors .....	139
8.1.7.1. Properties .....	140
8.1.8. New Expression .....	141
8.1.9. Function Expression .....	143
8.1.10. Function Calls .....	143
8.1.11. Postfix Expression .....	144
8.1.12. Unary Expression .....	145
8.1.13. Multiplicative Expression .....	146
8.1.14. Additive Expression .....	146
8.1.14.1. Type Inference .....	147
8.1.15. Bitwise Shift Expression .....	147
8.1.16. Relational Expression .....	148
8.1.17. Equality Expression .....	149
8.1.18. Binary Bitwise Expression .....	150
8.1.19. Binary Logical Expression .....	150
8.1.20. Conditional Expression .....	151
8.1.21. Assignment Expression .....	152
8.1.22. Comma Expression .....	153

8.2. ECMAScript 6 Expressions .....	154
8.2.1. The super Keyword .....	154
8.3. ECMAScript 7 Expressions .....	156
8.3.1. Await Expression .....	156
8.4. N4JS Specific Expressions .....	156
8.4.1. Class Expression .....	156
8.4.2. Cast (As) Expression .....	156
8.4.3. Semantics and Type Inference .....	156
8.5. Compile-Time Expressions .....	156
9. Statements .....	159
9.1. ECMAScript 5 Statements .....	159
9.1.1. Function or Field Accessor Bodies .....	159
9.1.2. Variable Statement .....	159
9.1.3. If Statement .....	161
9.1.4. Iteration Statements .....	161
9.1.5. Return Statement .....	162
9.1.6. With Statement .....	162
9.1.7. Switch Statement .....	163
9.1.8. Throw, Try, and Catch Statements .....	163
9.1.9. Debugger Statement .....	163
9.2. ECMAScript 6 Statements .....	164
9.2.1. Let .....	164
9.2.2. Const .....	164
9.2.3. <code>for ... of</code> statement .....	164
9.2.4. Import Statement .....	165
9.2.4.1. Dynamic Imports .....	169
9.2.4.2. Immutability of Imports .....	169
9.2.5. Export Statement .....	169
10. Annotations .....	171
10.1. Introduction .....	171
10.1.1. Syntax .....	171
10.1.2. Properties .....	171
10.1.3. Element-Specific Annotations .....	172
10.1.4. General Annotations .....	172
10.1.4.1. IDEBUG .....	172
10.1.5. Syntax .....	172
10.1.5.1. Semantics .....	172
10.1.5.2. Suppress Warnings .....	173
10.2. Declaration of Annotations .....	173
11. Extended Features .....	175
11.1. Array and Object Destructuring .....	175
11.1.1. Syntax .....	175
11.1.2. Semantics .....	175
11.2. Dependency Injection .....	176
11.2.1. DI Components and Injectors .....	177
11.2.1.1. DIComponent Relations .....	178
11.2.2. Binders and Bindings .....	179
11.2.3. Injection Points .....	181
11.2.3.1. Field Injection .....	181
11.2.3.2. Constructor Injection .....	181
11.2.3.3. Method Injection .....	182
11.2.4. N4JS DI Life Cycle and Scopes .....	182
11.2.4.1. Injection Cycles .....	183
11.2.4.2. Default Scope .....	184
11.2.4.3. Singleton Scope .....	184
11.2.4.4. Per Injection Chain Singleton .....	185
11.2.5. Validation of callsites targeting N4Injector methods .....	186
11.2.6. N4JS DI Annotations .....	186
11.2.6.1. N4JS DI <code>@GenerateInjector</code> .....	186
11.2.6.2. N4JS DI <code>@WithParentInjector</code> .....	186

11.2.6.3. N4JS DI @UseBinder .....	187
11.2.6.4. N4JS DI @Binder .....	187
11.2.6.5. N4JS DI @Bind .....	187
11.2.6.6. N4JS DI @Provides .....	187
11.2.6.7. N4JS DI @Inject .....	188
11.2.6.8. N4JS DI @Singleton .....	188
11.3. Test Support .....	188
11.4. Polyfill Definitions .....	188
11.4.1. Runtime Polyfill Definitions .....	189
11.4.2. Static Polyfill Definitions .....	190
11.4.3. Transpiling static polyfilled classes .....	191
12. Components .....	193
12.1. N4JS Platform Architecture .....	193
12.1.1. Overview .....	193
12.2. Component Types .....	194
12.2.1. Libraries .....	194
12.2.2. Runtime Environment and Runtime Libraries .....	194
12.2.3. Tests .....	194
12.2.4. Type Definitions .....	195
12.3. Component Content .....	195
12.4. Package.json File .....	195
12.4.1. Basic Properties .....	195
12.4.2. N4JS Properties .....	195
12.4.3. Constraints .....	198
12.5. Support for NPM Scopes .....	199
12.6. Component Dependencies .....	200
12.6.1. Runtime Environment Resolution .....	200
12.7. N4JS Type Definitions .....	201
12.7.1. Specify Type Definition .....	201
12.7.2. Name Conventions .....	201
12.7.3. Version Conventions .....	201
12.7.3.1. Define a New Type Definition Package .....	201
12.7.3.2. Using a Type Definition Package .....	202
12.7.3.3. Rational .....	202
12.8. Modules .....	202
12.9. Properties Files .....	203
12.9.1. Syntax .....	203
12.9.2. Constraints .....	203
12.10. API and Implementation Components .....	203
12.10.1. Execution of API and Implementation Components .....	204
12.11. API and Implementation With DI .....	204
13. PlainJS .....	207
13.1. Type Inference and Validation for Plain JS .....	207
13.2. External Declarations .....	207
13.2.1. Declaring externals .....	208
13.2.2. Instantiating external classes .....	208
13.2.3. Implementation of External Declarations .....	209
13.2.4. Example .....	209
13.3. Global Definitions .....	210
13.4. Runtime Definitions .....	211
13.5. Applying Polyfills .....	211
14. JSDoc .....	213
14.1. General N4JSDoc Features .....	213
14.1.1. Provided Inline Tags .....	213
14.1.1.1. @code .....	213
14.1.1.2. @link .....	213
14.2. N4JSDoc for User Projects .....	213
14.2.1. Standard Tags .....	213
14.2.1.1. @author .....	213
14.2.1.2. @param .....	213

14.2.1.3. @return .....	213
14.2.2. Test Related Tags .....	213
14.2.2.1. @testee .....	213
14.2.2.2. @testeeFromType .....	214
14.2.2.3. @testeeType and @testeeMember .....	214
14.2.2.4. @reqid in Tests .....	215
14.3. N4JSDoc for API and Implementation Projects .....	216
14.3.1. @apiNote .....	216
14.3.2. API Project Tags .....	216
14.3.2.1. @apiState .....	216
15. JSX .....	217
15.1. JSX Support .....	217
15.2. JSX Backend .....	219
16. Grammars .....	221
16.1. Type Expressions Grammar .....	221
16.2. N4JS Language Grammar .....	223
17. JSObjects .....	241
17.1. Object .....	241
17.2. String .....	241
17.3. Boolean .....	243
17.4. Number .....	243
17.4.1. Static Attributes .....	243
17.5. Function .....	243
17.6. Error .....	243
17.7. Array .....	244
17.8. Date .....	244
17.9. Math .....	246
17.9.1. Static Attributes .....	246
17.10. RegExp .....	247
17.11. JSON .....	247
18. N4JS Objects .....	249
18.1. Reflection Model .....	249
18.2. Error Types .....	254
18.2.1. N4ApiNotImplemented .....	254
A. Acronyms .....	255
B. Bibliography .....	257



---

# List of Figures

3.1. Forbidden Shadowing .....	15
4.1. Predefined Types Hierarchy .....	18
4.2. Cheat Sheet: Subtype Relation of Parameterized Types .....	26
4.3. Classifier and Constructor Type Subtype Relations .....	46
4.4. Constructors and prototypes for two classes A and B in ECMAScript 2015 (not N4JS!) .....	48
6.1. Function Variance Chart .....	106
8.1. Scopes .....	133
11.1. DI Basic Terms .....	176
11.2. Complex DIComponents Relations .....	179
11.3. Field Cycle .....	183
11.4. Ctor Field Cycle .....	183
11.5. Ctor Cycle .....	183
12.1. N4JS Component Overview .....	193
12.2. Content of a Component .....	194
12.3. Overview of API tests with DI .....	205
12.4. API tests with static DI .....	205
12.5. Types view and Instances view .....	206
13.1. External Class Implementation, Naming Convention .....	209
18.1. N4JS Reflection Classes .....	249



---

# List of Tables

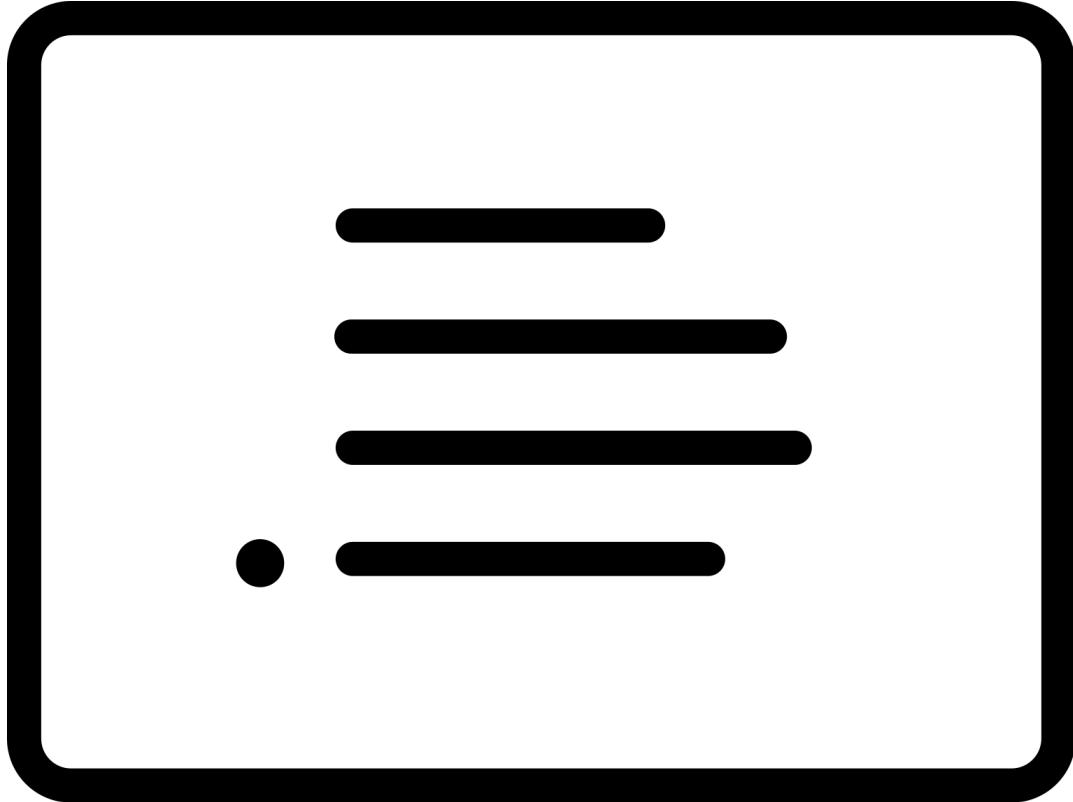
3.1. Type Access Control .....	8
3.2. Member Access Control .....	10
3.3. Different forms of module and type specifiers.....	13
4.1. Merged Members of Unions .....	43
4.2. Merged Members of Intersections .....	44
5.1. Extensibility of Types .....	61
5.2. Optional Fields .....	85
5.3. Consumption of methods .....	93
5.4. Available Fields of Structural Types .....	96
5.5. Member Availability in various Typing Scenarios .....	100
10.1. Element-Specific Annotations .....	172
11.1. DI No Bindings .....	184
11.2. DI Transitive Bindings .....	185
11.3. DI Re - Binding .....	185
11.4. DI Child Binding .....	185



# List of Examples

3.1. Implicit, allowed type references in N4JS .....	8
3.2. Synthesized Member Access Modifiers in N4JS .....	9
3.3. Type and Member Access Modifiers .....	11
3.4. Declaring type vs receiver type .....	11
3.5. Member Access and Type Access Interplay .....	12
4.1. Type Examples, Class Hierarchy .....	18
4.2. Type Annotation Syntax .....	19
4.3. Type Variable as Upper Bound .....	21
4.4. F bounded quantification .....	22
4.5. Type variables vs. existential types .....	22
4.6. Raw Types .....	24
4.7. Subtyping with parameterized types .....	25
4.8. Subtyping between different generic types .....	26
4.9. Existential Type .....	26
4.10. Upper and lower bound of parameterized types .....	27
4.11. Use declared interfaces for lower bounds .....	27
4.12. Composed types .....	35
4.13. Subtyping with union type .....	37
4.14. Superfluous elements in union type .....	38
4.15. Subtyping with intersection type .....	40
4.16. Static Polymorphism .....	46
4.17. Constructor of Abstract Class .....	46
4.18. Constructors and Prototypes .....	47
4.19. This keyword and type in instance and static context .....	49
4.20. This type in function-type-expression .....	51
4.21. Problems with this type and type arguments .....	51
4.22. Enumeration List .....	53
4.23. WebIDL example .....	54
5.1. Simple Class .....	60
5.2. Extend and implement .....	60
5.3. Simple Interfaces .....	63
5.4. Generic Type Definition and Usage as Type of Variable .....	64
5.5. Binding of type variables with multiple types .....	65
5.6. Use-site declaration of variance .....	66
5.7. Final Methods in Interfaces .....	72
5.8. Field name conflicts with structural member name .....	75
5.9. Anonymous Interface in Constructor .....	77
5.10. Spec Object and Subclasses .....	77
5.11. Superfluous Properties in @Spec Constructors .....	77
5.12. Interface fields that cannot be initialized via @Spec constructors .....	78
5.13. Covariant Constructors .....	80
5.14. Covariant Constructors in Interfaces .....	80
5.15. Getter and Setter .....	83
5.16. Static members of classes, inheritance and polymorphism .....	88
5.17. Static members in Java .....	89
5.18. Static members of interfaces .....	89
5.19. Method Consumption .....	93
5.20. Field and Field Initializer Consumption .....	94
5.21. Declaration Site Structural Typing .....	96
5.22. Nested Structural Typing Strategies .....	96
5.23. Effect of structural type modifiers on use-site .....	97
5.24. Use-Site Structural Typing .....	100
5.25. Structural types variable and instanceof operator .....	100
5.26. Subtype relationship between structural field typing .....	101
5.27. Setters with <code>@ProvidesInitializer</code> treated as optional .....	102
5.28. Access modifier in structural typing .....	102
5.29. Additional optional members in structural typing .....	104

5.30. Forbidden additional structural members on type variables .....	104
5.31. Use explicitly defined Interfaces .....	104
6.1. Function type conformance .....	107
6.2. Function Subtyping .....	107
6.3. Subtyping with function types .....	107
6.4. Bounded type variable at co-variant position in function type .....	109
6.5. Function Declaration with Type Annotation .....	112
6.6. Inference Of Function Expression's Signature .....	112
6.7. Two simple generator functions .....	114
6.8. Async as keyword and identifier .....	117
6.9. Type variables with async methods. ....	119
6.10. Promisifiable .....	120
7.1. Auto-Conversion .....	124
7.2. Type Check Example .....	126
7.3. Reflection with <code>N4Class</code> .....	127
7.4. Reflection with Built-In Types .....	127
7.5. <code>N4Class.of</code> .....	127
8.1. This in Unrestricted Mode .....	129
8.2. This in strict mode .....	130
8.3. This in N4JS mode .....	130
8.4. Effect of Nominal This Type .....	132
8.5. This and Function Declaration .....	132
8.6. This and Function Expressions .....	132
8.7. Array Type Inference .....	135
8.8. Scoping and linking .....	138
8.9. Parenthesized Expression Type Examples .....	139
8.10. Object as Map .....	141
8.11. Abstract classes and construction .....	142
8.12. Generic Method Invocation .....	144
8.13. Type of addition expression .....	147
8.14. <code>instanceof</code> with Interface .....	149
8.15. Type of Conditional Expressions .....	152
8.16. Comma Expression .....	153
8.17. Super Keyword .....	154
8.18. Super Call in Static Methods .....	155
9.1. Variable Statement .....	160
9.2. Import .....	166
9.3. Imports .....	167
9.4. Export .....	170
10.1. IDEBUG Example .....	172
11.1. Simple DIComponents Relation .....	178
11.2. Complex DIComponents Relations .....	179
11.3. Simple Field Injection .....	181
11.4. Static polyfill .....	191
12.1. Module Filters .....	196
12.2. A package.json file with N4JS-specific properties. ....	198
13.1. External Definitions and Their Implementations .....	209
13.2. External Classes .....	209
13.3. Structurally-typed external interfaces .....	210
13.4. <code>Object.observe</code> with Polyfill .....	211
14.1. <code>@testee</code> .....	213
14.2. Usage of <code>testeeFromType</code> .....	214
14.3. <code>testeeType</code> and <code>testeeMethod</code> .....	214
14.4. <code>testeeType</code> and <code>testeeMethod</code> with omitted constraints .....	215



# N4JS

Last Updated: 2018-11-06

**Authors:**

Jens von Pilgrim, Jakub Siberski, Mark-Oliver Reiser,  
Torsten Krämer, Ákos Kitta, Sebastian Zarnekow, Lorenzo Bettini, Jörg Reichert, Kristian Duske, Marcus Mews, Minh Quang Tran,  
Luca Beurer-Kellner

## Abstract

This document contains the N4JS Specification.

## Revision History

Date	Author	Description
2013-03-21	v.Pilgrim	initial commit
2016-03-13	v.Pilgrim	public release, version 0.3 (alpha)
2016-05-31	mor	support for definition-site variance, cf. <a href="#">Section 5.1.6, “Definition-Site Variance”</a>
2016-07-18	mor	rename manifest property <code>artifactId</code> to <code>projectId</code> ; remove <code> projectName</code>
2016-09-09	mor	add <code>@CovariantConstructor</code> , cf. <a href="#">Section 5.2.5.4, “Covariant Constructors”</a> , plus related adjustments
2016-11-09	bsmith	convert from <code>LaTeX</code> to AsciiDoc
2017-07-03	qtran	Describe semantics of optional fields in ternay expressions <a href="#">Section 5.2.6.5, “Optional Fields”</a>
2018-09-13	mmews	add subsection <a href="#">N4JS Type Definitions</a>

## Licence

This specification and the accompanying materials is made available under the terms of the Eclipse Public License v1.0 which accompanies this distribution, and is available at <http://www.eclipse.org/legal/epl-v10.html>

### Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC LICENSE ( **AGREEMENT** ). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

#### 1. DEFINITIONS

**Contribution means:** 1. in the case of the initial Contributor, the initial code and documentation distributed under this Agreement, and

2. in the case of each subsequent Contributor:

- a. changes to the Program, and
- b. additions to the Program;

where such changes and/or additions to the Program originate from and are distributed by that particular Contributor. A Contribution 'originates' from a Contributor if it was added to the Program by such Contributor itself or anyone acting on such Contributor's behalf. Contributions do not include additions to the Program which:

- i. are separate modules of software distributed in conjunction with the Program under their own license agreement, and
- ii. are not derivative works of the Program.

**Contributor** means any person or entity that distributes the Program.

**Licensed Patents** mean patent claims licensable by a Contributor which are necessarily infringed by the use or sale of its Contribution alone or when combined with the Program.

**Program** means the Contributions distributed in accordance with this Agreement.

**Recipient** means anyone who receives the Program under this Agreement, including all Contributors.

#### 2. GRANT OF RIGHTS

1. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free copyright license to reproduce, prepare derivative works of, publicly display, publicly perform, distribute and sublicense the Contribution of such Contributor, if any, and such derivative works, in source code and object code form.
2. Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.
3. Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.
4. Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

#### 3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

1. it complies with the terms and conditions of this Agreement; and

2. its license agreement:

- a. effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;
- b. effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;
- c. states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and
- d. states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

1. it must be made available under this Agreement; and
2. a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

#### 4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor (**Commercial Contributor**) hereby agrees to defend and indemnify every other Contributor (**Indemnified Contributor**) against any losses, damages and costs (collectively **Losses**) arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

#### 5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN **AS IS** BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

#### 6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

#### 7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions of this Agreement and does not cure such failure in a reasonable period of time after becoming aware of such noncompliance. If all Recipient's rights under this Agreement terminate, Recipient agrees to cease use and distribution of the Program as soon as reasonably practicable. However, Recipient's obligations under this Agreement and any licenses granted by Recipient relating to the Program shall continue and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in order to avoid inconsistency the Agreement is copyrighted and may only be modified in the following manner. The Agreement Steward reserves the right to publish new versions (including revisions) of this Agreement from time to time. No one other than the Agreement Steward has the right to modify this Agreement. The Eclipse Foundation is the initial Agreement Steward. The Eclipse Foundation may assign the responsibility to serve as the Agreement Steward to a suitable separate entity. Each new version of the Agreement will be given a distinguishing version number. The Program (including Contributions) may always be distributed subject to the version of the Agreement under which it was received. In addition, after a new version of the Agreement is published, Contributor may elect to distribute the Program (including its Contributions) under the new version. Except as expressly stated in Sections 2(a) and 2(b) above, Recipient receives no rights or licenses to the intellectual property of any Contributor under this Agreement, whether expressly, by implication, estoppel or otherwise. All rights in the Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the intellectual property laws of the United States of America. No party to this Agreement will bring a legal action under this Agreement more than one year after the cause of action arose. Each party waives its rights to a jury trial in any resulting litigation.



# Chapter 1. Introduction

This specification defines the N4JS language.

In general, the N4JS JavaScript dialect used is identical to the standard ECMAScript as defined in the 6th edition of ECMA-262, also known as ECMAScript 2015, referred to as [ECMA15a].

## 1.1. Notation

### 1.1.1. Grammar Notation

For the specification of the syntax and structure of elements, we use a slightly augmented similar to the grammar language of Xtext Grammar Language.

Similar to [ECMA11a], we define types with properties only for the purpose of explanation and usage within this specification. We use the Xtext notation style to assign values to meta-properties. Particularly, we use the Xtext notation for collection (`+ =`) and boolean (`? =`) values. These properties are written in italics. Enumerations are defined similar to Xtext. In order to allow the specification of default values, which are often defined by omitting the value, we always define the literal explicitly if it can be defined by the user.

The following lists informally defines the grammar:

<b>Terminal</b>	Terminals (or terminal strings) are enclosed in single quotes, e.g., <code>terminal</code> .
<b>Enumerations</b>	Rules which contain only terminals used as values for properties are marked with <code>enum</code> for enumeration.
<b>Properties</b>	Values of non-terminals, e.g., other rules, can be assigned to properties. The property name and the assignment are not part of the original syntax and only used for the meta description. E.g., <code>name=Identifier</code> .
<b>Collection Properties</b>	If a property is a collection, values are added to that list via <code>+ =</code> . E.g., <code>property+ = Value</code> .
<b>Boolean Properties</b>	Boolean properties are set to false by default, if the value (usually a terminal) is found, the boolean value is set to true. Often, the name of the property is similar to the terminal. E.g., <code>final? = 'final' ?</code> .

Properties of a non-terminal are sometimes listed again below the grammar. In that case, often pseudo properties are introduced which are derived from other properties and which are only used for simplification.

### 1.1.2. Type Judgments and Rules and Constraints Notation

#### 1.1.2.1. Typing Rules and Judgments

##### Rule

###### Definition: Rule

We use the common notation for rules such as type inference rules <sup>1</sup>, that is

$$\frac{\text{premises}}{\text{conclusion}} \quad \text{rule name}$$

*premises* is the rule's premises (e.g., the expression to be inferred), *conclusion* the result of the rule. *rulename* is an optional condition which may be omitted.

Both parts of the rule may contain multiple expressions, which are concatenated via 'and'.

For example, the following

$$\frac{P_1 \quad P_2 \quad P_3}{C}$$

can be read as

if  $P_1$ ,  $P_2$ , and  $P_3$  are all true, then  $C$  is true as well.

<sup>1</sup> A brief introduction can be found at <http://www.cs.cornell.edu/~ross/publications/mixedsite/tutorial.html>. In general, we refer the reader to [Pierce02a]

The following judgments (with relation symbols) are used:

**subtype** <

**type** : in which the left hand side is a declaration or expression, and the right hand side a type. We also use [...] as a function returning the (inferred) type of an expression.

**expectedTypeIn**  $\triangleleft$ : a relation with three arguments: *container*  $\triangleleft$  *expression:type* means, that *expression* is expected to be a subtype of *type* inside *container*

The following statement, for example, defines transitivity of subtypes (in a simplified manner):

$$\frac{\Gamma \vdash B < : A \quad \Gamma \vdash C < : B}{\Gamma \vdash C < : A}$$

$\vdash$  is the context containing (bound) type variables etc.,  $\vdash$  can be read as `entails`. Thus, the rule can be read as follows:

if the type B is a subtype of type A in context  $\Gamma$  (i.e. with constraints on type variables specified in  $\Gamma$ ), and if type C is a subtype of B, then C is also a subtype of A in context  $\Gamma$ .

In rules, we sometimes omit the environment if it is not needed. New information is sometimes added to the environment, in particular, substitutions (that is binding type variables to a type). The set of substitutions is written with  $\theta$  (theta). If new substitutions are explicitly added to that set, we write  $\theta(V \leftarrow T)$  ( $V$  is substituted with type  $T$ ). Often, these bindings are computed from a parameterized type reference which declares type arguments which are bound to the type variables of the generic declaration. In this case we simply write  $\theta(p)$ , in which  $p$  is the parameterized type declaration. As these new substitutions must become part of a (newly) created environment, we then usually write  $\Gamma \leftarrow \theta(p)$ . These substitutions are usually omitted.

### 1.1.2.2. Types of an Element

A variable or other typed element may be associated with three types:

1. Declared type: the type explicitly specified in the code, e.g., `var s: string`.
2. Inferred type: the type inferred by the type inferencer, e.g., `var s = "Hello"` infers the type of *s* to `string`. i.e.  $\Gamma \vdash s: string$  will be true, or `[s] < : string`. If an element is annotated with a type, i.e. it has a declared type, the inferred type will always be the declared type.
3. Actual type: the actual type of a variable during runtime. This type information is not available at compile time and ignored in this specification.

These types are not type declarations but type references, in fact, as they may be parameterized. For the sake of simplicity, we often omit the *Ref* suffix to shorten formulas. Consequently, we define the following properties and pseudo properties for typed elements such as variables:

**declaredTypeRef** The explicitly declared type, this is usually a real property of the construct. Not all elements allow the specification of a declared type, such as expressions.

**inferredTypeRef or [...]** This pseudo property is the inferred type computed by the type inferencer.

**typeRef** A pseudo property for elements with a *declaredType* property. It is similar to the inferred type, i.e. `e.type=[e]`

## 1.2. Auxiliary Functions

This section describes some auxiliary functions required for definition of type inference rules later on.

### 1.2.1. Binding

Binding an identifier (variable reference) to a variable declaration (or variable definition) is not part of this specification as this is standard ECMAScript functionality. However, some valid ECMAScript bindings are permitted due to visibility constraints.

#### Binding Relation

**Definition:** Binding Relation

We define a pseudo relation

`bind:VariableReference × VariableDeclaration`

which binds a reference, i.e. an identifier, to a declaration (e.g., variable declaration).

Binding of variable references to declaration is defined by ECMAScript already. Type references only occur in type expressions, how these are handled is explained in [Section 4.2, “Type Expressions”](#).

We usually omit this binding mechanism in most rules and use the reference similarly to the declaration or definition it is bound to. If a variable reference  $r$ , for example, is bound to a variable declaration  $D$ , i.e.  $\text{bind}(r, D)$ , we simply write  $r.type$  instead of  $\text{bind}(r, D).type$  to refer to the type expression (of the variable).<sup>2</sup>

A `DeclaredType` references the type declaration by its simple name that has been imported from a module specifier. We define the method  $\text{bind}$  for declared types as well:

#### Binding Relation of Types

**Definition:** Binding Relation of Types

We define a pseudo relation

$\text{bind}:\text{DeclaredType} \times \text{Class}|\text{Interface}|\text{Enum}$

which binds a type reference, i.e. a simple name, to the type declaration.

### 1.2.2. Merging Types

In some cases we have to merge types, e.g., types of a union type or item types of an array. For that purpose, we define a method  $\text{merge}$  as follows.

#### Merge Function

**Definition:** Merge Function

We define a pseudo function

$\text{merge}:\text{Type} \times \dots \times \text{Type} \Rightarrow \text{P}(\text{Type})$

The idea of this function is to remove duplicates. For example; if a union type contains two type expressions  $te_1$  and  $te_k$ , and if  $\tau(te_1) = \tau(te_2)$ , then  $\text{merge}(\tau(te_1), \tau(te_2))$  contains only one element. The order of the elements is lost, however.

#### 1.2.2.1. Logic Formulas

In general, we use a pragmatic mixture of pseudo code, predicate logic, and OCL. Within constraints (also within the inference rules), the properties defined in the grammar are used.

In some rules, it is necessary to type the rule variables. Instead of explicitly checking the metatype (via  $\mu(X) = :MetaType$ ), we precede the variable with the type, that is:  $\text{MetaType}X$ .

Instead of “type casting” elements, often properties are simply accessed. If an element does not define that element, it is either assumed to be false or null by default.

If a property  $p$  is optional and not set, we write  $p = \text{null}$  to test its absence. Note that  $p = \text{null}$  is different from  $p = \text{Null}$ , as the latter refers to the null type. Non-terminals may implicitly be subclasses. In that case, the concrete non-terminal, or type, of a property may be subject for a test in a constraint.

### 1.2.3. Symbols and Font Convention

Variables and their properties are printed in italic when used in formulas (such as rules). A dot-notation is used for member access, e.g.  $v.name$ . Also defined functions are printed in italic, e.g.,  $acc(r, D)$ . Properties which define sets are usually ordered and we assume 0-indexed access to elements, the index subscripted, e.g.,  $v.methods_i$ .

We use the following symbols and font conventions:

$\wedge, \vee, \oplus, \neg$

Logical and, or, exclusive or (xor), and not.

<sup>2</sup> One can interpret this similar to delegate methods, that is, instead of writing `r.binding().getType()`, a method `r.getType()\{return binding().getType();\}` is defined.

$=, \Leftrightarrow, \text{if}\dots, \text{then}\dots, \text{else}\dots$	Logical implication, if and only if, and if-then-else.
$\text{true}, \text{false}, \text{null}, \emptyset$	Boolean true, boolean false, null (i.e., not specified, e.g., $v.sup =$ means that there are no $sup$ (super class) specified), empty set.
$\in, \notin, \cup, \cap,  \chi $	Element of, not an element of, union set, intersection set, cardinality of set $x$ .
$P(X)$	Power set of $X$ , i.e. $P(X) = \{U : U \subseteq X\}$ .
$\exists, \nexists, \forall$	Exists, not exists, for all; we write $\exists x, \dots, z : P(x, \dots, z)$ and say "there exists $x, \dots, z$ such that predicate $P$ is true".
	Note that $\nexists x : P(x) \Leftrightarrow \forall x : \neg P(x)$ .
$\mu(\dots)$	(mu) read "metatype of"; metatype of a variable or property, e.g., $\text{if}\dots \mu(x) = :Class \text{then}\dots \text{else}\dots$
$\bar{x}$	Sequence of elements $x_1, \dots, x_n$ . E.g., if we want to define a constraint that the owner of a members of a class $C$ is the class, we simply write
	$C.\overline{\text{members}}.owner = C$
	instead of
	$\forall m \in C.\text{members}:m.owner = C$
	or even more complicated with index variables.

Sequences are 1-based, e.g., a sequence  $s$  with length  $|s| = n$ , has elements  $s_1, \dots, s_n$ .

# Chapter 2. Grammar

## 2.1. Lexical Conventions

As a super language on top of ECMAScript, the same lexical conventions are supported as described in [ECMA11a(p.S7)] within strict mode. Some further constraints are defined, however, restricting certain constructs. These constraints are described in the following.

### 2.1.1. Identifier Names and Identifiers

Cf. [ECMA11a(p.S7.6)], [ECMA11a(p.S11.1.2, p.p.63)] and [ECMA11a(p.S01.2, p.p.51ff)].

As a reminder, identifiers are defined as follows in the ECMAScript specification:

```
IdentifierName: IdentifierStart* IdentifierPart;
IdentifierStart : UnicodeLetter | '_';
\ UnicodeEscapeSequence
```

N4JS supports a limited form of computed-names for member declarations:

```
N4JSPropertyComputedName:
  '[' (SymbolLiteralComputedName | StringLiteralComputedName) ']'
;

SymbolLiteralComputedName: N4JSIdentifier `.` N4JSIdentifier ;
StringLiteralComputedName: STRING ;
```

As can be seen, a computed-name must be either

- a symbol reference, e.g., `Symbol.iterator`
- a string literal, i.e., a compile time known constant. This notation is useful when interoperating with libraries that define members whose names contain special characters (e.g., a field name starting with commercial-at)

In N4JS, identifiers are further constrained in order to avoid ambiguities and to make code more readable. Some of these constraints will lead to errors, others only to warnings. They do not apply for identifiers declared in definitions file (n4jsd) in order to enable declaration of external entities.

#### N4JS Identifier Restrictions

##### Requirement: IDE-1: N4JS Identifier Restrictions (ver. 1)

. If the following constraints do not hold, errors are created. . Leading `$` (dollar sign) character is prohibited for any variable name such as fields, variables, types functions and methods. . Leading `_` (underscore) character is not allowed for identifying any functions or methods.

#### N4JS identifier recommendations

##### Requirement: IDE-2: N4JS identifier recommendations (ver. 1)

. If the following constraints do not hold, warnings are created. . Variable names should, in general, be constructed from the 26 ASCII upper and lower case alphabetic letters (a..z, A..Z), from the 10 decimal digits (0..9) and from the `_` (underscore). Although the usage of the international characters are allowed (according to the ECMAScript specification) it is discouraged because these characters may not be read or understood well in every circumstance <sup>1</sup>.

##### 1. Type (and Type Variable) Identifiers

```
TypeIdentifier: [_A-Z][_a-zA-Z0-9]*
TypeVariableIdentifier: [_A-Z][_a-zA-Z0-9]*
```

##### 2. Package Identifiers

```
PackageIdentifier: [_a-z][_a-zA-Z0-9]*; // i.e. the folder names, must not end with .
```

<sup>1</sup> <http://javascript.crockford.com/code.html>

### 3. Member Identifiers and Enum Literals

```
InstanceFieldIdentifier: [_a-z][_a-zA-Z0-9]*
StaticFieldIdentifier: [_A-Z][_A-Z0-9]*([_A-Z0-9]+)*
EnumLiteral: [_A-Z][_A-Z0-9]*([_A-Z0-9]+)*
```

### 4. Variable and Parameter Names

```
VariableIdentifier: [_a-zA-Z0-9]*
##ParameterIdentifier: [_a-z][_a-zA-Z0-9]*
```

### 5. Methods

```
MethodIdentifier: [_a-zA-Z0-9]*;
```

### 6. Annotations

```
AnnotationIdentifier: [_A-Z][_a-zA-Z0-9]*
```

The following rules describe how fully qualified names of elements are created. Note that these fully qualified names cannot be used in N4JS directly. Though they may be shown in error messages etc. to identify elements.

```
TypeIdentifier: [A-Z][a-zA-Z0-9]*;
PackageIdentifier: [a-z][a-zA-Z0-9]*;
FQNTType: (PackageIdentifier '.')+ TypeIdentifier;
```

## 2.1.2. This Keyword

Cf. [[ECMA11a\(p.S11.1.1, p.p.63\)](#)]

## 2.1.3. Regular Expression Literals

Cf. [[ECMA11a\(p.S7.8.5\)](#)]

## 2.1.4. Automatic Semicolon Insertion

Cf. [[ECMA11a\(p.S7.9\)](#)]

ASI is supported by the parser, however warnings are issued.

## 2.1.5. JSDoc

JSDoc are comments similar to JavaDoc in Java for documenting types, functions and members. There is no semantic information expressed in JSDoc, that is, the behavior of a program must not change if all the JSDoc is removed. The JSDoc tags and overall syntax is a mixture of tags defined by the [Google Closure Compiler](#), Java's [JavaDoc](#) tool and N4-specific tags.

JSDoc comments are multiline comments, starting with `/***` (instead of simple multiline comments, starting with `/*`).

```
MultiLineComment: '/*' MultiLineCommentChars? '*/' // from ECMA Script specification
JSDoc: '/*' MultiLineCommentChars? '*/'
```

In general, JSDoc comments are placed directly before the annotated language element. In some cases, this is slightly different, such as for method parameters, for example, where it is then explicitly specified.

The content of JSDoc comments will be covered in more detail in upcoming chapters. For documentation purposes, multi- and single-line descriptions are used in several constructs.

```
MLVALUE: ([^@]+[^\\n]+)+;
SLVALUE: ([^\\n]+);
```

**MLVALUE** short for `multi-line value`. This is usually only used for the general description of types or members.

**SLVALUE** short for `single-line value`. This is a description which ends at the end of a line. It is usually used in combination with other tags, e.g., to further describe a parameter of a method.

# Chapter 3. Names

*Visibility* defines the scope in which a declaration is visible, that is in which context references can be bound to the declaration. *Access control* defines the extent to which types and members are accessible beyond their immediate context. Access control may, therefore, restrict the visibility of a declaration by limiting its scope.

*Extensibility* refers to whether a given type can be subtyped, or in the case of members, whether they can be overridden. Access control is a prerequisite for extensibility which is further explained in [Section 5.1, “N4JS Specific Classifiers”](#)

## 3.1. Access Control

Types from one project may or may not be made accessible to another project. Likewise, members from a given type may or may not be made accessible to members existing outside that type. For example, if a developer writes an application which uses a library, which types within that library can the application see? Given a type that is set as visible, which members of that type can the application see?

Accessing a type or member actually means that a reference is bound to a declaration with the same identifier.

We distinguish the following contexts from which an element is accessed as follows:

1. **Module or type**: access from elements in the same module or type.
2. **Subtype**: access from a subtype.
3. **Project**: access from the same project.
4. **Vendor**: access from different project of the same vendor.
5. **World**: access from anything else.

Accessibility is defined by modifiers on types and members, e.g `public`, `protected`, `project`, `private`, via the `export` statement, and by the `@Internal` annotation. Extensibility is defined by the `@Final` annotation respectively.

## 3.2. Accessibility of Types, Top-Level Variables and Function Declarations

We define types (classes, interfaces, enums) whereby each type has members (fields and methods, depending on the kind of type). When we define a type, we need to define whether it is visible only for the specifying module, project or whether that type should be accessible from outside of that project.

The same is true for variable declarations and function declarations defined as top-level elements of a module.

The following type access modifiers are supported by N4JS:

```
enum TypeAccessModifier:    project  
                           | public;
```

If a type is not exported, its visibility is private. If a type has declared visibility `public`, it may additionally be marked as internal via the annotation `@Internal`. Thus, we have the following set of type access modifiers:

```
TAM = private project public@Internal public
```

That is, in N4JS, only the type access modifiers and are available. The redundant `project` modifier serves only documentation purpose and can be synthesized if the `export` modifier is present.

All other modifiers used here are synthesized as shown in the next example:

### Synthesized Type Access Modifiers in N4JS

```
class C0 {}                      // private  
export class C1 {}                // project  
export project class C1 {}        // project  
export @Internal public class C1 {} // public@Internal  
export public class C2 {}          // public  
  
var v0;                          // private  
export var v1;                    // project  
export project var v1;            // project  
export @Internal public var v3;   // public@Internal  
export public var v2;              // public
```

```

function f0() {}           // private
export function f1() {}    // project
export project function f1() {} // project
export @Internal public function f3() {} // public@Internal
export public function f2() {} // public

```

The access control levels are defined as listed in [Table 3.1, “Type Access Control”](#).

**Table 3.1. Type Access Control**

Type Access				
Modifier	Module	Project	Vendor	World
<code>private</code>	yes	no	no	no
<code>project</code>	yes	yes	no	no
<code>public@Internal</code>	yes	yes	yes	no
<code>public</code>	yes	yes	yes	yes

`TAM` is a totally ordered set:

`private < project < public@Internal < public`

### Type Access Modifiers

#### Requirement: IDE-3: Type Access Modifiers (ver. 1)

The following constraints for type access modifiers for a given type  $T$  (which may be a classifier declaration, a function or a variable) must hold:

- It is an error if  $T$  is not exported but defined as `project`, `public` or `public@Internal`.
- It is an error if an annotation `@Internal` is present on a module private or `project` visible type.
- The type modifier for all built-in ECMAScript types is `public`.
- The *default modifier* for user declared exported declarations is `project`. That is, this modifier is assumed if no modifier is explicitly specified.

### Type Accessibility $\alpha_T$

#### Definition: Type Accessibility $\alpha_T$

The function  $\alpha_T : TypeReference \times TypeDeclaration \Rightarrow Boolean$  computes whether a given type, (top-level) variable or function reference can access the declaration that it references.  $\alpha_T$  is defined with [Table 3.1, “Type Access Control”](#).

Formally, we define  $\alpha_T$  for a given reference  $r$  and a module top level variable, function or type declaration  $D$  as follows <sup>1</sup>:

$$\begin{aligned} D.acc = \text{public} &\quad \alpha_T(r, D) \\ D.acc = \text{public}@Internal &\quad r.vendor = D.vendor \quad \alpha_T(r, D) \\ D.acc = \text{project} &\quad r.project = D.project \quad \alpha_T(r, D) \\ D.acc = \text{private} &\quad r.module = D.module \quad \alpha_T(r, D) \end{aligned}$$

If the type of the arguments is clear from the context, we simply write  $\alpha(r, D)$  instead of  $\alpha_T(r, D)$ .

Accessibility for types is only checked for types that manifest themselves in the concrete syntax of the N4JS file. Types that do not have to be written to concrete syntax may be used even if they are generally not accessible. This is illustrated by [Example 3.1, “Implicit, allowed type references in N4JS”](#):

#### Example 3.1. Implicit, allowed type references in N4JS

```
export public class D {
```

<sup>1</sup> See for definitions of metatype properties.

```

public takeC(): C { ... }
public acceptC(c: C): void { ... }
}
/* private */ class C {}

var d: D = new D()
d.acceptC( d.takeC() )

```

### 3.2.1. Accessibility of Members

Accessibility at the member level is only applicable when the type itself is accessible. If you cannot access the type, you cannot access any of its members. Note that inherited members (from an interface or class) become members of a class. For example, if `B extends A`, and if `A` is not accessible to some client `C` but `B` is, then the members of `A` are indirectly accessible to `C` in so far as they are accessed via `B`. This is true in particular for interfaces, as their properties are possibly merged into the consuming class (cf. [Section 5.2.8.2, “Implementation of Members”](#)).

The following member access modifiers are supported by N4JS:

```

enum MemberAccessModifier:    private
                             | project
                             | protected
                             | public;

```

The modifiers `protected` and `public` may be annotated with `@Internal`. Thus, we can define the following set of member access modifiers:

```

MAM = {
private, protected@Internal, protected,
        project, public@Internal, public
}

```

`protected@Internal` and `public@Internal` are synthesized tags and were introduced as shorthand notation for the `@Internal` annotation together with `protected` or `public` access modifiers. The `project` modifier is the default one and it can be omitted. As with the type access modifiers, not all member access modifiers are available in N4JS. Instead, they are synthesized from different construct as shown in the next example.

#### Example 3.2. Synthesized Member Access Modifiers in N4JS

```

export @Internal public class C {

    private f0;                      // private
    f1;                            // project
    project f2;                     // project
    @Internal protected f3;          // protected@Internal
    protected f4;                     // protected
    @Internal public f5;             // public@Internal
    public f6;                       // public

    private m0() {}                 // private
    m1() {}                         // project
    project m2() {}                // project
    @Internal protected m3() {}     // protected@Internal
    protected m4() {}                // protected
    @Internal public m5() {}         // public@Internal
    public m6() {}                  // public
}

```

`MAM` does not define a totally ordered set. However, its subset

`MAM \ {public@Internal}`

is a totally ordered set<sup>2</sup>:

`private < project < protected@Internal < protected < public`

[Table 3.2, “Member Access Control”](#) shows which members are accessible from where.

<sup>2</sup> That is, for application developers not providing a library or a public API available to other vendors, member access modifiers behave almost similar to modifiers known from Java.

Table 3.2. Member Access Control

Access Modifier	Inside Module	Inside Project	Vendor	Vendor Subtypes	Other Projects	Everywhere
<code>private</code>	yes	no	no	no	no	no
<code>project</code>	yes	yes	no	no	no	no
<code>protected@Internal</code>	yes	yes	yes	no	no	no
<code>protected</code>	yes	yes	yes	no	yes	no
<code>public@Internal</code>	yes	yes	yes	yes	no	no
<code>public</code>	yes	yes	yes	yes	yes	yes

### Type and Member Accessibility Relation

**Definition:** Type and Member Accessibility Relation

We define the relation

$$= : TAM \times MAM$$

as follows:

$$= ::= \{ (\text{private}, \text{private}), (\text{project}, \text{project}), \\ (\text{public}@Internal, \text{public}@Internal), (\text{public}, \text{public}) \}$$

We further define the relation  $\geq : TAM \times MAM$  as follows:

$$\exists mam' \in MAM : tam = mam' \wedge mam' \geq mam \\ tam \geq mam$$

Less, greater than etc. are defined accordingly.

### Member Accessibility

**Definition:** Member Accessibility

The function

$$\alpha_m : MemberReference \times MemberDeclaration \Rightarrow Boolean$$

computes if a given reference can access the member declaration that it references.

Note that  $\alpha_m$  and  $bind$  are different functions. A reference can only bind to a declaration if it can access the declaration. However,  $bind$  requires more condition to work (correct metatypes, no shadowing etc).

Formally, we define  $\alpha_m$  for a given reference  $r$  and member declaration  $M$  as follows:<sup>3</sup> <sup>4</sup>

$$\begin{aligned} & M.acc = \text{public} \\ & \alpha_m(r, M) \\ \underline{r.vendor = M.vendor \quad M.acc = \text{public}@Internal} & \alpha_m(r, M) \\ & r.owner \in r.receiver.super^* \quad M.acc = \text{protected} \\ & \alpha_m(r, M) \\ \underline{r.owner \in r.receiver.super^* \quad r.vendor = M.vendor \quad M.acc = \text{protected}@Internal} & \alpha_m(r, M) \\ & r.project = M.project \quad M.acc = \text{project} \\ & \alpha_m(r, M) \\ & r.module = r.module \quad M.acc = \text{private} \\ & \alpha_m(r, M) \end{aligned}$$

<sup>3</sup> See Section 5.1, “N4JS Specific Classifiers” for definitions of metatype properties. Note that  $r.receiver$  always refers to a type declaration in the context of an expression as the receiver type of  $r$ . The declaring type of the member declaration is considered to be the receiver type of the member reference rather than the type that originally declares the member declaration.

<sup>4</sup> Note the Java-like access restriction for members of visibility `protected` or `protected@Internal` to code that is responsible for the implementation of that object.  
[Gosling15a(p.S6.6.2, p.p.166)]

If the type of the arguments is clear from the context, we simply write  $\alpha(r, M)$  instead of  $\alpha_m(r, M)$ .

Although private members are accessible inside a module, it is not possible to redefine (override etc.) these members (see Section 5.2.8, “Redefinition of Members”).

### Default Member Access Modifiers

#### Requirement: IDE-4: Default Member Access Modifiers (ver. 1)

The following constraints for member access modifiers must hold:

1. The *default modifier* for members of user-declared classes is `project`.
2. The *default modifier* for members of interfaces is the same as the visibility of the interface itself, except for private interfaces. For private interfaces, the default modifier for members is `project`.
3. The modifier for enum literals is always `public`.
4. Private members of a classifier are visible and accessible within a module, i.e. you can access the private method of a class, for instance, when the use of the class as receiver is in the same module where the class has been defined. In case of inheritance, private members are visible if the host (e.g. the class) is in the same module as the provider (the extended class). This also means that abstract members of a class are allowed to be defined private as they may be overridden within a module.

### Example 3.3. Type and Member Access Modifiers

```
export project interface I {
    project foo();
}

// This interface may be used publicly, but since the inherited method foo() is project visible only,
// it is not possible to implement that interface in other projects.
export public interface J extends I {
}

// Since the visibility of foo is set to public here, it is possible to implement this interface in other projects.
export public interface K extends I {
    @Override public foo();
}

// Since foo is private, it is not possible to subclass the class in other modules. Still, it
// is possible to use it in other projects.
// XPECT noerrors -->
export public abstract class C {
    private abstract foo();

    public static C instance() {
        // return some default instance
        ...
    }
}
```

As demonstrated in the following snippet, class `C` can be used but not subclassed in other modules:

```
import C from "C"

// XPECT errors --> "Cannot extend class C: cannot implement one or more non-accessible abstract members: method C.foo." at
// "C"
export public abstract class Sub extends C {
}

// XPECT noerrors -->
var c: C = C.instance();
```

Members of non-visible types are, in general, not visible for a client. Members may become visible, however, if they are accessed via a visible type which inherits these members. The following examples demonstrate two different scenarios:

### Example 3.4. Declaring type vs receiver type

It is especially noteworthy that the declaring type of a member is generally not considered for the accessibility of that member but only the receiver type is relevant.

```
class Base {
```

```

public m(b: Base): void {}
}
export public class ApiType extends Base {
}

import * as N from "Base";

var t = new N.ApiType();
// member can be accessed although type Base is not exported:
t.m(t);

```

The property access to the member `m` is valid because it fulfills the constraints for accessibility. The receiver of the property access is `t` of type `ApiType`. That type is exported and accessible. Therefore, the inherited member `m` is also considered valid since it is also defined `public`.

This rule allows for defining a common functionality in module or project visible types that becomes accessible via exported, visible subtypes.

#### Example 3.5. Member Access and Type Access Interplay

The following example demonstrates the behavior when non-visible types are used as return types. In this case, all the members of the non-visible types are not accessible, even if they have a public access modifier.

```

class A {
    foo(): void{}
}
export public class C {
    public getHidden(): A { return new A(); }
}

import * as N from "A"

class Client {
    f(): void {
        var c = new N.C();
        // XPECT noerrors --> Getting an instance the hidden type is possible
        var hidden = c.getHidden();
        // XPECT errors --> "The method foo is not visible." at "foo"
        hidden.foo();
    }
}

```

### 3.2.2. Valid Names

For identifier and property names, the same constraints as in ECMAScript [ECMA11a(p.S7.6)] [ECMA11a(p.S7.6.1.2)] [ECMA11a(p.S11.6)] are applied.

Identifier names in N4JS are defined similar to [ECMA11a(p.S11.6)], making it possible to even use reserved words (keywords etc.). For some element types, errors or warnings are issued in order to prevent problems when using these names.

#### Forbidden Identifier Names in N4JS

##### Requirement: IDE-5: Forbidden Identifier Names in N4JS (ver. 1)

In N4JS mode, errors are generated in the following cases:

1. A name of a type equals
  - a. an access modifier
  - b. `set` or `get`
  - c. an ECMAScript keyword
  - d. a boolean literal
  - e. the name of a base type
2. The name of a function or function expression equals (but not the method)
  - a. an ECMAScript keyword

- b. a reserved future ECMAScript word

### Undesired Identifier Names in N4JS

**Requirement: IDE-6: Undesired Identifier Names in N4JS** (ver. 1)

In N4JS mode, warnings are generated in the following cases:

1. The name of a member (of a non external type)
  - a. equals the name of a base type <sup>5</sup> but the type of the variable is different from that type
  - b. is not static nor const but starts with an upper case letter
2. The name of a non-external N4 types (class, interface, enum) starts with a lower case letter
3. The name of a variable (incl. formal parameter or catch variable and fields)
  - a. equals an N4JS keyword
  - b. equals the name of a base type but the type of the variable is different from that type
  - c. is not const but starts with an upper case letter

### 3.2.3. Qualified Names

In N4JS source code, types can only be referenced using their simple name. There is no such thing as a fully-qualified type name in N4JS or ECMAScript. Types are uniquely identified by their simple name, maybe together with an import and the module specifier given there. Clashes between simple names of imported type and locally declared types can be resolved by importing the type under an alias.

In some cases, however, we need to define references to types or even members. For example, if we want to reference certain members in JSDoc comments or for unambiguous error messages. For this reason, we formally define qualified names even if they cannot occur in source code.

Table 3.3, “Different forms of module and type specifiers.” shows the different names of a given type `C`, defined in a module `M.n4js`, defined in a package `p` of a project `MyProject`.

Simple type names are used throughout N4JS code in order to refer to types. The different forms of module specifiers are only used in import declarations in the string following the `from` keyword.

Table 3.3. Different forms of module and type specifiers.

Name	Example
Simple Type Name	<code>C</code>
(Plain) Module Specifier	<code>p/M</code>
Complete Module Specifier	<code>MyProject/p/M</code>
Complete Type Specifier	<code>MyProject/p/M.C</code>

### 3.2.4. Name Duplicates

There might be cases where two (or more) scopes created by different entities with the same (simple) name overlap. Those situations can be referred to as shadowing, hiding, or obscuring. While they are not the same, many of those cases are not allowed in N4JS. For simplicity we refer to them all as shadowing or duplication (see below). Rule of thumb is that N4JS allows everything that is allowed in JavaScript StrictMode.

#### 3.2.4.1. Lexical Environment

N4JS handles scopes similar to ECMAScript, so that function scope is applied to variables declared with `var` (and parameters), and block scope for variables is declared with `let` or `const`. In general, ECMAScript defines *Lexical Environments* as a specification type used to define the association of Identifiers to specific variables and functions based upon the lexical nesting structure of ECMAScript code [ECMA11a(p.10.2)].

**Elements that introduce lexical environments:** `FunctionDefinition`, `VariableDeclaration`, `CatchBlock`, `WithStatement`, `ImportDeclaration`

<sup>5</sup> `string`, `boolean`, `number`, `any`, `null`

**N4JS specific declarations:** `N4ClassDeclaration`, `N4InterfaceDeclaration`, `N4EnumDeclaration`, `N4MethodDeclaration`.

Additionally, a built-in lexical environment that defines global scope exists for every `Script`.

Since N4JS is extended (and a bit more strict) JS strict mode, *Object environment records* created by `WithStatement` are not taken into account when resolving duplicates. This applies to both N4JS mode and JS strict mode. In unrestricted JS the `WithStatement` is allowed but duplicates are not validated.



In case of names introduced by `ImportDeclaration`s only `NamedImportSpecifiers`s are taken into account (their import name or its alias if available). `WildcardImportSpecifiers`s are not taken into account. Potential optimizations by compiler or user annotation are also not currently taken into account during analysis.

### 3.2.4.2. Duplicates and Shadowing

#### Shadowing Overriding Duplicates

**Definition:** [Shadowing Overriding Duplicates](#)

Two elements with the same name declared in the same lexical environment (cf. [ECMA11a(p.S10.2.2.1)]) are called *duplicates*. An element defined in an environment *shadows* all elements with the same name in outer environments.

In class hierarchies, a member with the same name as a member defined in a supertype is said to override the latter. Overriding is discussed in [Section 5.2.8, “Redefinition of Members”](#).

For the following constraints, we make the following assumptions:

- Names of function expressions or declarations are handles similar to locally declared elements in the function. Function declarations are additionally declaring a name in their outer scope.
- The implicit formal parameter `arguments` is treated similar to declared formal parameters.
- Formal parameters are defined in the lexical environment of a function, that is, they are defined in the same lexical environment as local `var`-variables or other declarations in that function.
- The “global” environment contains objects globally defined by the execution environment.

#### Forbidden Duplicates

**Requirement: IDE-7: Forbidden Duplicates** (ver. 1)

There must be no two elements defined in the same lexical environment with the same name, that is, there must be no duplicates.

#### Forbidden Shadowing

**Requirement: IDE-8: Forbidden Shadowing** (ver. 1)

In general, shadowing is allowed in N4JS. But it is not allowed in the following cases:

- No element defined in the standard global scope must be shadowed.
- There must be no function shadowing another function.
- Elements defined in catch blocks must not shadow elements defined all parent non-catch-block environments.

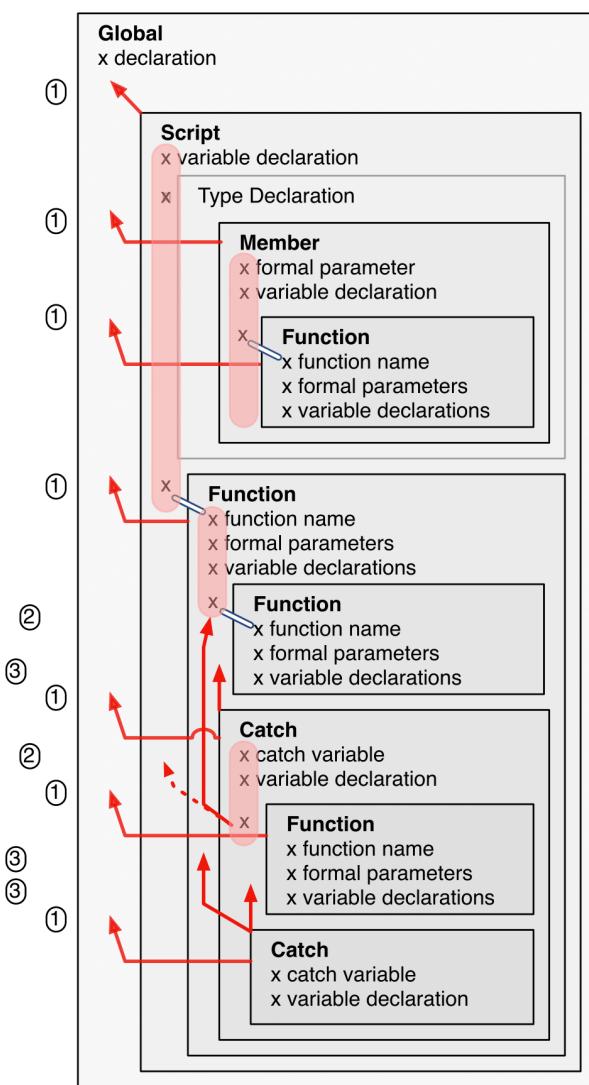
#### Forbidden Names

**Requirement: IDE-9: Forbidden Names** (ver. 1)

- In the script environment, it is not allowed to use the name ‘`arguments`’.<sup>6</sup>

+ [Figure 3.1, “Forbidden Shadowing”](#) shows nested lexical environments with named elements declared inside (all named `x` here), the forbidden cases are marked with arrows (the numbers at the left side refer to the numbers in [Req-IDE-8]).

<sup>6</sup> This conflicts with the implicit parameter arguments introduced by the transpiler when wrapping the script/module into a definition function.

**Figure 3.1. Forbidden Shadowing**

Rationale:

- We expect only few named nested functions. Since this is expected to be a rare case, no shadowing should occur there as this is maybe not expected by the programmer.
- It is typical that nested environments define local variables. In particular helper variables (such as `i: number` or `s: string`) are expected to be used quite often. Since this is a typical case, we allow shadowing for local variables.
- Function declarations may shadow type declarations. However, both entities are to be handled completely differently, so that an error will occur if the shadowing is ignored by the programmer anyway.



# Chapter 4. Types

## 4.1. Overview

N4JS is essentially ECMAScript with the inclusion of types. In the following sections we will describe how types are defined and used in N4JS.

Besides standard JavaScript types, the following metatypes are introduced:

- Classifiers, that is class or interface (see [Chapter 5, Classifiers](#))
- Enum

Classifiers, methods and functions may be declared generic.

Types are related to each other by the subtype relation.

### Subtype Relation

#### Definition: Subtype Relation

We use *subtype* for the general subtype relation or type conformance.

In nominal typing,  $T < : S$  means that  $S$  is a (transitive) supertype of  $T$ . Generally in structural typing, this means that  $T$  conforms to  $S$ .  $< :$  is defined transitive reflexive by default.

We write  $<$  to refer to the transitive non-reflexive relation, that is  $T < S \Leftrightarrow T < : S \wedge T \neq S$

Whether nominal or structural typing is used depends on the declaration of the type or the reference. This is explained further in [Section 5.3, “Structural Typing”](#).

For convenience reasons, we sometimes revert the operator, that is  $T < : S \Leftrightarrow S :> T$ . We write  $T \not< : S$  if  $T$  is not type conforming to  $S$ . (cf. [\[Gosling12a\(p.S4.10\)\]](#))

Join and meet are defined as follows:

### Join and Meet

#### Definition: Join and Meet

A type  $J$  is called a *join* (or least common supertype, ) of a pair of types  $S$  and  $T$ , written  $S \vee T = J$ , if

$$\begin{aligned} S &< : J \\ T &< : J \\ \forall L.(S < : L) \wedge (T < : L) \rightarrow J &< : L \end{aligned}$$

Similarly, we say that a type  $M$  is a *meet* (or greatest common subtype, ) of  $S$  and  $T$ , written  $S \wedge T = M$ , if

$$\begin{aligned} M &< : S \\ M &< : T \\ \forall L.(L < : S) \wedge (L < : T) \rightarrow L &< : M \end{aligned}$$

Note that this declarative definition needs to be specified in detail for special cases, such as union and intersection types. Usually, the union type of two types is also the join.

[Figure 4.1, “Predefined Types Hierarchy”](#) summarizes all predefined types, that is primitive and built-in ECMAScript and N4JS types. Specific rules for the subtype relation are defined in the following sections. This type hierarchy shows `any` and `undefined` as the top and bottom type (cf. [\[Pierce02a\(p.15.4\)\]](#)) We define these types here explicitly:

### Top and Bottom Type

#### Definition: Top and Bottom Type

We call *Top* the top type, if for all types  $T$  the relation  $T < : \text{Top}$  is true. We call *Bot* the bottom type, if for all types  $T$  the relation  $\text{Bot} < : T$  is true. In N4JS,  $\text{Top} = \text{any}$ , the bottom type  $\text{Bot} = \text{undefined}$ .

`null` is almost similar to `Bot`, except that it is not a subtype of `undefined`.

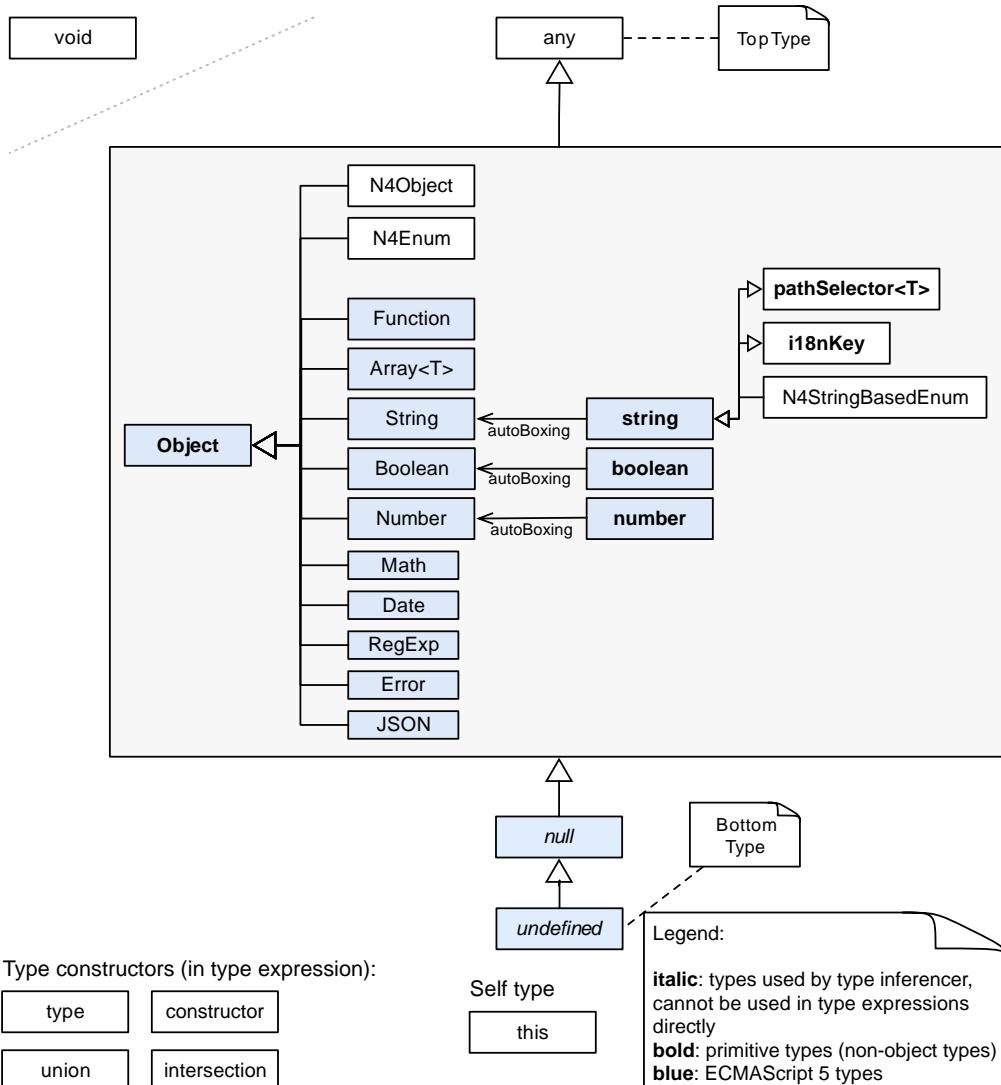


Figure 4.1. Predefined Types Hierarchy

For every primitive type there is a corresponding built-in type as defined in [ECMA11a], e.g. `string` and `String`. There is no inheritance supported for primitive types and built-in types – these types are final.

Although the diagram shows inheritance between `void` and `undefined`, this relationship is only semantic: `void` is a refinement of `undefined` from a type system viewpoint. The same applies to the relation of `Object` as well as the subtypes shown for `string` and `String`.

#### Example 4.1. Type Examples, Class Hierarchy

In the following examples, we assume the following classes to be given:

```
// C <: B <: A
class A{}
class B extends A{}
class C extends B{}

// independent types X, Y, and Z
class X{} class Y{} class Z{}

// interfaces I, I1 <: I, I2 <: I, I3
interface I
interface I1 extends I {}
```

```

interface I2 extends I {}
interface I3 {}

// class implementing the interfaces
class H1 implements I1{}
class H12 implements I1,I2{}
class H23 implements I2,I3{}

// a generic class with getter (contra-variance) and setter (co-variance)
class G<T> {
    get(): T;
    set(x: T): void;
}

```

## 4.2. Type Expressions

In contrast to ECMAScript, N4JS defines static types. Aside from simple type references, type expressions may be used to specify the type of variables.

### 4.2.1. Syntax

The listing [EBNF Type Expression Grammar](#) summarizes the type expression grammar. Depending on the context, not all constructs are allowed. For example, the variadic modifier is only allowed for function parameters.

References to user-declared types are expressed via `ParameterizedTypeRef`. This is also true for non-generic types, as the type arguments are optional. See [Section 4.4.3, “Parameterized Types”](#) for details on that reference.

For qualified names and type reference names, see [Section 3.2.3, “Qualified Names”](#)

The type expressions are usually added to parameter, field, or variable declarations as a suffix, separated with colon ( `:` ). The same is true for function, method, getter or setter return types. Exceptions in the cases of object literals or destructuring are explained later on.

#### Example 4.2. Type Annotation Syntax

The following two listings show the very same code and type annotations are provided on the left hand side. For simplicity, `string` is always used as type expression.<sup>1</sup>

```

var x: string;
var s: string = "Hello";
function f(p: string): string {
    return p;
}
class C {
    f: string;
    s: string = "Hello";
    m(p: string): string {
        return p;
    }
    get x(): string {
        return this.f;
    }
    set x(v: string) {
        this.f = v;
    }
}

```

```

var x;
var s = "Hello";
function f(p) {
    return p;
}
class C {
    f;
    s = "Hello";
    m(p) {
        return p;
    }
    get x() {
        return this.f;
    }
    set x(v) {
        this.f = v;
    }
}

```

The code on the right hand side is almost all valid ECMAScript 2015, with the exception of field declarations in the class. These are moved into the constructor by the N4JS transpiler.

### 4.2.2. Properties

Besides the properties indirectly defined by the grammar, the following pseudo properties are used for type expressions:

Properties of `TypeExpression`:

`var` If true, variable of that type is variadic. This is only allowed for parameters. Default value: `false`.

<sup>1</sup> In the N4JS IDE, type annotations are highlighted differently than ordinary code.

<code>opt</code>	If true, variable of that type is optional. This is only allowed for parameters and return types. This actually means that the type $T$ actually is a union type of <code>Undef   T</code> . Default value: <code>false</code> .
<code>optvar</code>	$optvar = var v opt$ , reflect the facts that a variadic parameter is also optional (as its cardinality is $[0 \dots n]$ ).
<code>entity</code>	Pseudo property referencing the variable declaration (or expression) which <code>owns</code> the type expression.

### 4.2.3. Semantics

The ECMAScript types `undefined` and `null` are also supported. These types cannot be referenced directly, however. Note that `void` and `undefined` are almost similar. Actually, the inferred type of a types element with declared type of `void` will be `undefined`. The difference between void and undefined is that an element of type void can never have another type, while an element of type undefined may be assigned a value later on and thus become a different type. `void` is only used for function and method return types.

Note that not any type reference is allowed in any context. Variables or formal parameters must not be declared `void` or union types must not be declared dynamic, for example. These constraints are explained in the following section.

The types mentioned above are described in detail in the next sections. They are hierarchically defined and the following list displays all possible types. Note that all types are actually references to types. A type variable can only be used in some cases, e.g., the variable has to be visible in the given scope.

### ECMAScript Types

<b>Predefined Type</b>	Predefined types, such as String, Number, or Object; and .
<b>Array Type</b>	<a href="#">Section 4.7.4, “Array Object Type”</a> .
<b>Function Type</b>	Described in <a href="#">Chapter 6, Functions</a> , <a href="#">Section 6.1, “Function Type”</a> .
<b>Any Type</b>	<a href="#">Section 4.6.1, “Any Type”</a> .

### N4Types

<b>Declared Type</b>	(Unparameterized) Reference to defined class <a href="#">Section 5.1.3, “Classes”</a> or enum <a href="#">Section 4.13, “Enums”</a> .
<b>Parameterized Type</b>	Parameterized reference to defined generic class or interface; <a href="#">Section 4.4.3, “Parameterized Types”</a> .
<b>This Type</b>	<a href="#">Section 4.12, “This Type”</a> .
<b>Constructor and Type Type</b>	Class type, that is the meta class of a defined class or interface, <a href="#">Section 4.11, “Constructor and Classifier Type”</a> .
<b>Union Types</b>	Union of types, <a href="#">Section 4.10.1, “Union Type”</a> .
<b>Type Variable</b>	Type variable, <a href="#">Section 4.4.2, “Type Variables”</a> .

Type expressions are used to explicitly declare the type of a variable, parameter and return type of a function or method, fields (and object literal properties).

### 4.3. Type Inference

If no type is explicitly declared, it is inferred based on the given context, as in the expected type of expressions or function parameters, for example. The type inference rules are described in the remainder of this specification.

#### Default Type

**Definition:** Default Type

In N4JS mode , if no type is explicitly specified and if no type information can be inferred, `any` is assumed as the default type.

In JS mode, the default type is `any+`.

Once the type of a variable is either declared or inferred, it is not supposed to be changed.

Given the following example.

#### Variable type is not changeable

```
var x: any;
```

```
x = 42;
x=5; // error: any is not a subtype of number.
```

Type of `x` is declared as `any` in line 1. Although a number is assigned to `x` in line 2, the type of `x` is not changed. Thus an error is issued in line 3 because the type of `x` is still `any`.

At the moment, N4JS does not support type guards or, more general, effect system (cf. [Nielson99a]).

## 4.4. Generic and Parameterized Types

Some notes on terminology:

**Type Parameter vs. Type Argument** A type parameter is a declaration containing type variables. A type argument is a binding of a type parameter to a concrete type or to another type parameter. Binding to another type parameter can further restrict the bounds of the type parameter.

This is similar to function declarations (with formal parameters) and function calls (with arguments).

### 4.4.1. Generic Types

A class declaration or interface declaration with type parameters declares a generic type. A generic type declares a family of types. The type parameters have to be bound with type arguments when referencing a generic type.

### 4.4.2. Type Variables

A type variable is an identifier used as a type in the context of a generic class definition, generic interface definition or generic method definition. A type variable is declared in a type parameter as follows.

#### Syntax

```
TypeVariable:
  (declaredCovariant?='out' | declaredContravariant?='in')?
  name=IdentifierOrThis ('extends' declaredUpperBound=TypeRef)?
;
```

#### Example 4.3. Type Variable as Upper Bound

Note that type variables are also interpreted as types. Thus, the upper bound of a type variable may be a type variable as shown in the following snippet:

```
class G<T> {
  <X extends T> foo(x: X): void { }
}
```

#### Properties

A type parameter defines a type variable, which type may be constrained with an upper bound.

Properties of `TypeVariable`:

`name` Type variable, as type variable contains only an identifier, we use type parameter instead of type variable (and vice versa) if the correct element is clear from the context.

`declaredUpperBound` Upper bound of the concrete type being bound to this type variable, i.e. a super class.

#### Semantics

##### Type Variable Semantics

**Requirement: IDE-10: Type Variable Semantics (ver. 1)**

1. Enum is not a valid metatype in `declaredUpperBounds`.
2. Wildcards are not valid in `declaredUpperBounds`.
3. Primitives are not valid in `declaredUpperBounds`.
4. Type variables are valid in `declaredUpperBounds`.

A type variable can be used in any type expression contained in the generic class, generic interface, or generic function / method definition.

**Example 4.4. F bounded quantification**

Using a type variable in the upper bound reference may lead to recursive definition.

```
class Chain<C extends Chain<C, T>, T> {
    next() : C { return null; }
    m() : T { return null; }
}
```

**Type Inference**

In many cases, type variables are not directly used in subtype relations as they are substituted with the concrete types specified by some type arguments. In these cases, the ordinary subtype rules apply without change. However, there are other cases in which type variables cannot be substituted:

1. Inside a generic declaration.
2. If the generic type is used as raw type.
3. If a generic function / method is called without type arguments and without the possibility to infer the type from the context.

In these cases, an unbound type variable may appear on one or both sides of a subtype relation and we require subtype rules that take type variables into account.

It is important to note that while type variables may have a declared upper bound, they cannot be simply replaced with that upper bound and treated like existential types. The following example illustrates this:

**Example 4.5. Type variables vs. existential types**

```
class A {}
class B extends A {}
class C extends B {}

class G<T> {}

class X<T extends A, S extends B> {

    m(): void {

        // plain type variables:
        var t: T;
        var s: S;

        t = s; // ERROR: "S is not a subtype of T." at "s" ①

        // existential types:
        var ga: G<? extends A>;
        var gb: G<? extends B>;

        ga = gb; ②
    }
}
```

- ① Even though the upper bound of `s` is a subtype of `T`'s upper bound (since `B < : A`), we cannot infer that `s` is a subtype of `T`, because there are valid concrete bindings for which this would not be true: for example, if `T` were bound to `C` and `s` to `B`.
- ② This differs from existential types (see `ga` and `gb` and line 21):  
`G<? extends B> < : G<? extends A>`.

We thus have to define subtype rules for type variables, taking the declared upper bound into account. If we have a subtype relation in which a type variable appears on one or both sides, we distinguish the following cases:

1. If we have type variables on both sides: the result is true if and only if there is the identical type variable on both sides.
2. If we have a type variable on the left side and no type variable on the right side:  
the result is true if and only if the type variable on the left has one or more declared upper bounds.  
`intersection(left.declaredUpperBounds) < : right`  
This is the case for

`(T extends B) < : A`

in which `T` is an unbound type variable and `A, B` two classes with `B < : A`.

3. In all other cases the result is false.

This includes cases such as

$$B < : (T \text{ extends } A)$$

which is always false, even if  $B < : A$  or

$$(T \text{ extends } A) < : (S \text{ extends } B)$$

which is always false, even if  $A = B$ .

We thus obtain the following definition:

### Subtype Relation for Type Variables

#### Definition: Subtype Relation for Type Variables

For two types  $T, S$  of which at least one is a type variable, we define

- if both  $T$  and  $S$  are type variables:

$$\frac{T=S}{T < : S}$$

- if  $T$  is a type variable and  $S$  is not:

$$\frac{T.\text{declaredUpperBounds.size} > 0 \wedge \forall t \in T.\text{declaredUpperBounds}: t < : S}{T < : S}$$

### 4.4.3. Parameterized Types

References to generic types (cf. [Section 5.1.3, “Classes”](#)) can be parameterized with type arguments. A type reference with type arguments is called parameterized type.

#### Syntax

```

ParameterizedTypeRef:
    ParameterizedTypeRefNominal | ParameterizedTypeRefStructural;

ParameterizedTypeRefNominal:
    declaredType=[Type|TypeReferenceName]
    (=> '<' typeArgs+=TypeArgument (',' typeArgs+=TypeArgument)* '>')?;

ParameterizedTypeRefStructural:
    definedTypingStrategy=TypingStrategyUseSiteOperator
    declaredType=[Type|TypeReferenceName]
    (=> '<' typeArgs+=TypeArgument (',' typeArgs+=TypeArgument)* '>')?
    ('with' TStructMemberList)?;

TypeArgument returns TypeArgument:
    Wildcard | TypeRef;

Wildcard returns Wildcard:
    '?'
    (
        'extends' declaredUpperBound=TypeRef
        | 'super' declaredLowerBound=TypeRef
    )?
;
```

#### Properties

Properties of parameterized type references (nominal or structural):

<code>declaredType</code>	Referenced type by type reference name (either the simple name or a qualified name, e.g. in case of namespace imports).
<code>typeArgs</code>	The type arguments, may be empty.
<code>definedTypingStrategy</code>	Typing strategy, by default nominal, see <a href="#">Section 5.3, “Structural Typing”</a> for details
<code>structuralMembers</code>	in case of structural typing, reference can add additional members to the structural type, see <a href="#">Section 5.3, “Structural Typing”</a> for details.

#### Pseudo Properties:

<code>importSpec</code>	The <code>ImportSpecifier</code> , may be null if this is a local type reference. Note that this may be a <code>NamedImportSpecifier</code> . See <a href="#">Section 9.2.4, “Import Statement”</a> for details for details.
<code>moduleWideName</code>	Returns simple name of type, that is either the simple name as declared, or the alias in case of an imported type with alias in the import statement.

## Semantics

The main purpose of a parameterized type reference is to simply refer to the declared type. If the declared type is a generic type, the parameterized type references defines a *substitution* of the type parameters of a generic type with actual type arguments. A type argument can either be a concrete type, a wildcard or a type variable declared in the surrounding generic declaration. The actual type arguments must conform to the type parameters so that code referencing the generic type parameters is still valid.

### Parameterized Types

#### Requirement: IDE-11: Parameterized Types (ver. 1)

For a given parameterized type reference  $R$  with  $G = R.declaredType$ , the following constraints must hold:

- The actual type arguments must conform to the type parameters, that is:

$$\begin{aligned} |G.typeParams| &= |R.typeArgs| \\ \wedge \forall i, 0 < i < |R.typeArgs| \quad &[R.typeArgs_i] < : [R.typeParams_i] \end{aligned}$$

We define type erasure similar to Java [[Gosling12a\(p.S4.6\)](#)] as 'mapping from types (possibly including parameterized types and type variables) to types (that are never parameterized types or type variables)'. We write  $T^0$  for the erasure of type  $T$ .<sup>2</sup>

### Parameterized Type

#### Definition: Parameterized Type

A parameterized type reference  $R$  defines a parameterized type  $T$ , in which all type parameters of  $R.declaredType$  are substituted with the actual values of the type arguments. We call the type  $T^0$ , in which all type parameters of  $R.declaredType$  are ignored, the *raw type* or *erasure* of  $T$ .

We define for types in general:

- The erasure  $G^0$  of a parameterized type  $G<T_1, \dots, T_n>$  is simply  $G$ .
- The erasure of a type variable is the erasure of its upper bound.
- The erasure of any other type is the type itself.

This concept of type erasure is purely defined for specification purposes. It is not to be confused with the `real` type erasure which takes place at runtime, in which almost no types (except primitive types) are available.

That is, the type reference in `var G<string> gs;` actually defines a type `G<string>`, so that `[gs]=G<string>`. It may reference a type defined by a class declaration `class G<T>`. It is important that the type `G<string>` is different from `G<T>`.

If a parameterized type reference  $R$  has no type arguments, then it is similar to the declared type. That is,  $[R]=T=R.declaredType$  if (and only if)  $|R.typeArgs|=0$ .

In the following, we do not distinguish between parameter type reference and parameter type – they are both two sides of the same coin.

#### Example 4.6. Raw Types

In Java, due to backward compatibility (generics were only introduced in Java 1.5), it is possible to use raw types in which we refer to a generic type without specifying any type arguments. This is not possible in N4JS, as there is no unique interpretation of the type in that case as shown in the following example. Given the following declarations:

```
class A{}
```

<sup>2</sup> The notation  $|T|$  used in [[Gosling12a](#)] conflicts with the notation of cardinality of sets, which we use in case of union or intersection types for types as well. The notation used here is inspired by [[Crary02a](#)], in which a mapping is defined between a typed language  $\lambda$  to an untyped language  $\lambda^0$ .

```
class B extends A{}
class G<T extends A> { t: T; }
var g: G;
```

In this case, variable `g` refers to the *raw type* `G`. This is forbidden in N4JS, because two interpretations are possible:

1. `g` is of type `G<? extends A>`
2. `g` is of type `G<A>`

In the first case, an existential type would be created, and `g.t = new A();` must fail.

In the second case, `g = new G<B>();` must fail.

In Java, both assignments work with raw types, which is not really safe. To avoid problems due to different interpretations, usage of raw types is not allowed in N4JS.<sup>3</sup>

Calls to generic functions and methods can also be parameterized, this is described in [Section 8.1.10, “Function Calls”](#). Note that invocation of generic functions or methods does not need to be parameterized.

## Type Conformance

### Definition: Type Conformance

We define type conformance for non-primitive type references as follows:

- For two non-parameterized types  $T^0$  and  $S^0$ ,

$$\frac{S^0 \in T^0.sup^* \cup T^0.interfaces^*}{T^0 < : S^0}$$

- For two parameterized types  $T < T_1, \dots, T_n >$  and  $S < S_1, \dots, S_m >$

$$\frac{T^0 < : S^0}{T < : S} \quad (n=0 \vee m=0 \vee (n=m \rightarrow \forall i: \\ T_i.upperBound < : S_i.upperBound \\ \wedge T_i.lowerBound > S_i.lowerBound))\}$$

### Example 4.7. Subtyping with parameterized types

Let classes A, B, and C are defined as in the chapter beginning ( $C < : B < : A$ ). The following subtype relations are evaluated as indicated:

```
G<A> <: G<B>           -> false
G<B> <: G<A>           -> false
G<A> <: G<A>           -> true
G<A> <: G<?>           -> true
G<? extends A> <: G<? extends A> -> true
G<? super A> <: G<? super A> -> true
G<? extends A> <: G<? extends B> -> false
G<? extends B> <: G<? extends A> -> true
G<? super A> <: G<? super B> -> true
G<? super B> <: G<? super A> -> false
G<? extends A> <: G<A> -> false
G<A> <: G<? extends A> -> true
G<? super A> <: G<A> -> false
G<A> <: G<? super A> -> true
G<? super A> <: G<? extends A> -> false
G<? extends A> <: G<? super A> -> false
G<?> <: G<? super A> -> false
G<? super A> <: G<?> -> true
G<?> <: G<? extends A> -> false
G<? extends A> <: G<?> -> true
```

The figure [Figure 4.2, “Cheat Sheet: Subtype Relation of Parameterized Types”](#) shows the subtype relations of parameterized types (of a single generic type), which can be used as a cheat sheet.

<sup>3</sup> Although raw type usage is prohibited, the N4JS validator interprets raw types according to the first case, which may lead to consequential errors.

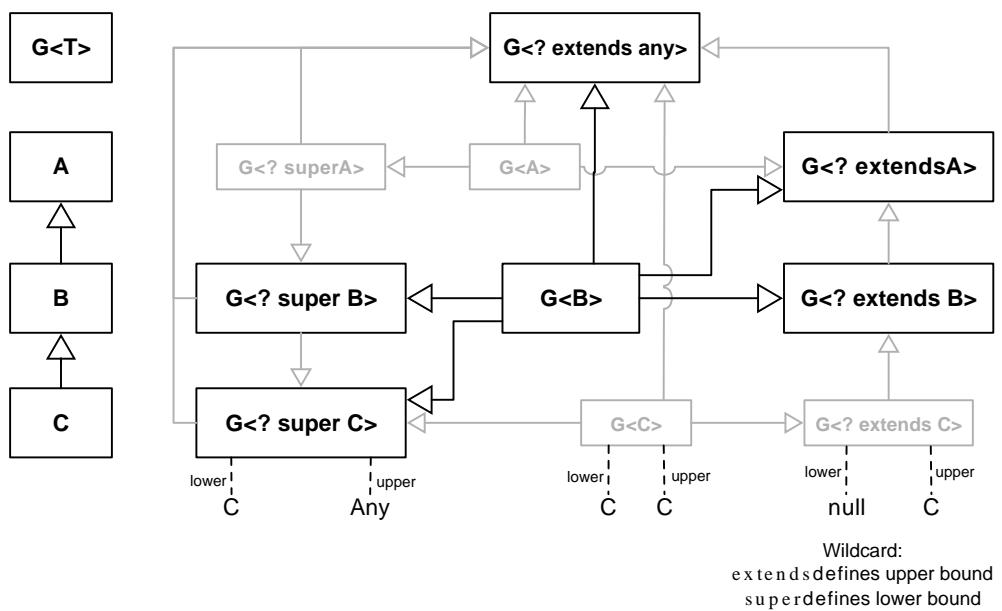


Figure 4.2. Cheat Sheet: Subtype Relation of Parameterized Types

**Example 4.8. Subtyping between different generic types**

Let classes  $G$  and  $H$  be two generic classes where:

```
class G<T> {}
class H<T> extends G<T> {}
```

Given a simple, non-parameterized class  $A$ , the following subtype relations are evaluated as indicated:

```
G<A> <: G<A>           -> true
H<A> <: G<A>           -> true
G<A> <: H<A>           -> false
```

**Type Inference**

Type inference for parameterized types uses the concept of existential types (in Java, a slightly modified version called capture conversion is implemented).

The general concept for checking type conformance and inferring types for generic and parameterized types is described in [Igarashi01a] for *Featherweight Java with Generics*.

The concept of existential types with wildcard capture (a special kind of existential type) is published in [Torgersen05a], further developed in [Cameron08b] (further developed in [Cameron09a] [Summers10a], also see [Wehr08a] for a similar approach). The key feature of the Java generic wildcard handling is called capture conversion, described in [Gosling12a(p.S5.1.10)]. However, there are some slight differences to Java 6 and 7, only with Java 8 similar results can be expected. All these papers include formal proofs of certain aspects, however even these paper lack proof of other aspect

The idea is quite simple: All unbound wildcards are replaced with freshly created new types <sup>4</sup>, fulfilling the constraints defined by the wildcard's upper and lower bound. These newly created types are then handled similar to real types during type inference and type conformance validation.

**Example 4.9. Existential Type**

The inferred type of a variable declared as

```
var x: G<? extends A>;
```

that is the parameterized type, is an existential type  $E_1$ , which is a subtype of  $A$ . If you have another variable declared as

```
var y: G<? extends A>;
```

another type  $E_2$  is created, which is also a subtype of  $A$ . Note that  $E_1 \neq E_2$ ! Assuming typical setter or getter in  $G$ , e.g. `set(T t)` and `T get()`, the following code snippet will produce an error:

<sup>4</sup> in the Java 8 spec and compiler, they are called type variables, which are types as well

```
y.set(x.get())
```

This is no surprise, as `x.get()` actually returns a type  $E_1$ , which is not a subtype of  $E_2$ .

The upper and lower bound declarations are, of course, still available during type inference for these existential types. This enables the type inferencer to calculate the join and meet of parameterized types as well.

### Join of Parameterized Types

#### Requirement: IDE-12: Join of Parameterized Types (ver. 1)

The join of two parameterized types  $G < T_1, \dots, T_n >$  and  $H < S_1, \dots, S_m >$  is the join of the raw types, this join is then parameterized with the join of the upper bounds of type arguments and the meet of the lower bounds of the type arguments.

For all type rules, we assume that the upper and lower bounds of a non-generic type, including type variables, simply equal the type itself, that is for a given type  $T$ , the following constraints hold:

$$\text{upper}(T) = \text{lower}(T) = T$$

#### Example 4.10. Upper and lower bound of parameterized types

Assuming the given classes listed above, the following upper and lower bounds are expected:

```
G<A>           -> upperBound = lowerBound = A
G<? extends A> -> lowerBound = null, upperBound = A
G<? super A>   -> lowerBound = A, upperBound = any
G<?>           -> lowerBound = null, upperBound = any
```

This leads to the following expected subtype relations:

```
(? extends A) <: A  -> true
(? super A) <: A    -> false
A <: (? extends A) -> false
A <: (? super A)   -> true
```

Note that there is a slight difference to Java: In N4JS it is not possible to use a generic type in a raw fashion, that is to say without specifying any type arguments. In Java, this is possible due to backwards compatibility with early Java versions in which no generics were supported.

In case an upper bound of a type variable shall consist only of a few members, it seems convenient to use additional structural members, like on interface I2 in the example [Example 4.11, “Use declared interfaces for lower bounds”](#) below. However, type variables must not be constrained using structural types (see constraint [Req-IDE-76]). Hence, the recommended solution is to use an explicitly declared interface that uses definition site structural typing for these constraints as an upper bound (see interface in [Example 4.11, “Use declared interfaces for lower bounds”](#)).

#### Example 4.11. Use declared interfaces for lower bounds

```
interface I1<T extends any with {prop : int}> { // error
}

interface ~J {
  prop : int;
}
interface I2<T extends J> {
```

## 4.5. Primitive ECMAScript Types

N4JS provides the same basic types as ECMAScript [[ECMA11a\(p.p.28\)](#)].



In ECMAScript, basic types come in two flavors: as primitive types [[ECMA11a\(p.S8Types, p.p.28\)](#)] and as Objects [[ECMA11a\(p.S15, p.p.102\)](#)]. In N4JS, primitive types are written with lower cases, object types with first case capitalized. For example, `string` is the primitive ECMAScript string type, while `String` is an object.

The following ECMAScript primitive types are supported, they are written with lower case letters::

- `undefined` [ECMA11a(p.S8.3)]; cannot be used in type expression, see void below.
- `null` [ECMA11a(p.S8.3)]; cannot be used in type expression
- `boolean` [ECMA11a(p.S8.3)]
- `string` [ECMA11a(p.S8.4)]
- `number` [ECMA11a(p.S8.5)]

Although Object is a primitive type in [ECMA11a(p.S8.5)], it is interpreted here as an object type and described in Section 4.7.2, "Object Type".

Please note that primitive types are values (= no objects) so they have no properties and you cannot inherit from them.

#### 4.5.1. Undefined Type

As a built-in type, the type `undefined` cannot be declared explicitly by the user by means of a type expression. Note in ECMAScript there are three distinct use cases of `undefined`:

- `undefined` as type (as used here)
- `undefined` as value (the only value of the undefined type)
- `undefined` is a property of the global object with undefined (value) as initial value. Since ECMAScript 5 it is not allowed to reassign this property but this is not enforced by all ECMAScript/JavaScript engines.

The type `undefined` will be inferred to false in a boolean expression. It is important to note that something that is not assigned to a value is `undefined` but not `null`.

The type `undefined` is a subtype of all types. That is,

$$\Gamma \vdash \text{undefined} < : T$$

is an axiom and true for all types  $T$ .

Whenever an expression  $E$  has an inferred type of `undefined`, which means it will always evaluate to value `undefined` at runtime, a warning is shown, unless  $E$  is ...

- a `void` expression
- the direct child expression of a `void` expression,
- the direct child expression of an expression statement,
- the `undefined` literal (i.e. the literal representing the `undefined` value),
- the `this` literal.

#### 4.5.2. Null Type

The `null` type cannot be declared explicitly by the user. Only the keyword `null` is inferred to type `null`.

##### Semantics

In contrast to `undefined`, it expresses the intentional absence of a value.

The `null` type can be assigned to any other type. That is, the type `null` is a subtype of all other types except `undefined`:

$$\frac{\text{right} \neq \text{undefined}}{\Gamma \vdash \text{null left} < : \text{Type right}}$$

Please note that

- `null==undefined` evaluates to `true`
- `null==!=undefined` evaluates to `false`
- `typeof null` evaluates to `object`

Only the `null` keyword is inferred to type `null`. If `null` is assigned to a variable, the type of the variable is not changed. This is true, in particular, for variable declarations. For example in

```
var x = null;
```

the type of variable `x` is inferred to `any` (cf. [Section 9.1.2, “Variable Statement”](#)).

The type `null` will be inferred to false in a boolean expression.

The call `typeof null` will return ‘object’.

### 4.5.3. Primitive Boolean Type

Represents a logical entity having two values, true and false.

Please note that a boolean primitive is coerced to a number in a comparison operation so that

Source	Result
<code>var a = true; console.log(a == 1)</code>	prints true
<code>var b = false; console.log(b == 0)</code>	prints true

#### Semantics

The type `boolean` is subtype of `any`:

`boolean < : any`

Variables of type `boolean` can be auto-converted (coerced) to `Boolean`, as described in [Section 7.1, “Autoboxing and Coercing”](#).

### 4.5.4. Primitive String Type

A finite sequence of zero or more 16-bit unsigned integer values (elements). Each element is considered to be a single UTF-16 code unit.

Also string as primitive type has no properties, you can access the properties available on the object `String` as string will be coerced to `String` on the fly but just for that property call, the original variable keeps its type:

```
var a = "MyString"
console.log(typeof a) // string
console.log(a.length) // 8
console.log(typeof a) // string
```

You can handle a primitive `string` like an object type `String` but with these exceptions:

- `typeof "MyString"` is ‘string’ but `typeof new String("MyString")` is ‘object’
- `"MyString" instanceof String` or `instanceof Object` will return `false`, for `new String("MyString")` both checks evaluate to `true`
- `console.log(eval("2+2"))` returns `4`, `console.log(eval(new String("2+2")))` returns string `"2+2"`

This marks a difference to Java. In JavaScript, Unicode escape sequences are never interpreted as a special character.

#### Semantics

The `string` type is a subtype of `any`:

`string < : any`

It is supertype of the N4JS primitive type `pathselector`, `typeName` and `i18nKey`. [Section 4.6.4, “Primitive Pathselector and I18nKey”](#)

However, variables of type `string` can be auto-converted (coerced) to `String`, as described in [Section 7.1, “Autoboxing and Coercing”](#).

### 4.5.5. Primitive Number Type

In ECMAScript numbers are usually 64-bit floating point numbers. For details see [[ECMA11a\(p.8.5\)](#)]. The prefix `0` indicates that the number is octal-based and the prefix `0x` marks it as hexadecimal-based.

`Nan` can be produced by e.g. ‘`0 / 0`’ or ‘`1 - x`’. `typeof NaN` will return `number`.

## Semantics

The type `number` is subtype of `any`:

$\text{number} < : \text{any}$

However, variables of type `number` can be auto-converted (coerced) to `Number`, as described in [Integer Literals](#).

## 4.5.6. Primitive Type int

Actually ECMAScript defines an internal type `int32`. A number of this type is returned by the binary or operation using zero as operand, e.g. ECMAScript's internal type `int32` can be represented in N4JS by a built-in primitive type called `int`. For details on how numeric literals map to types `number` and `int`, refer to [Section 8.1.3.1, “Integer Literals”](#).



for the time being, built-in type `int` is synonymous to type `number`. This means one can be assigned to the other and a value declared to be of type `int` may actually be a 64-bit floating point number.<sup>5</sup>

## 4.5.7. Primitive Symbol Type

The primitive type `symbol` is directly as in ECMAScript 6. Support for symbols is kept to a minimum in N4JS. While this primitive type can be used without any restrictions, the only value of this type available in N4JS is the built-in symbol `Symbol.iterator`. Other built-in symbols from ECMAScript 6 and the creation of new symbols are not supported. For more details, see [Section 4.7.9, “Symbol”](#).

## 4.6. Primitive N4JS Types

Additionally to the primitive ECMAScript types, the following N4JS-specific primitive types are supported:

<code>any</code>	enables ECMAScript-like untyped variable declarations
<code>void</code>	almost similar to undefined, except it can be used as a return type of functions and methods
<code>unknown</code>	inferred in case of a type inference error
<code>pathSelector&lt;T&gt;</code> ,	subtypes of string
<code>i18nKey</code>	

### 4.6.1. Any Type

Any type is the default type of all variables for without a type declaration. It has no properties. A value of any other type can be assigned to a variable of type `any`, but a variable declared `any` can only be assigned to another variable declared with the type `any`.

#### 4.6.1.1. Semantics

`any` is supertype of all other types. That is,

$\Gamma \vdash \text{Typeleft} < : \text{any}$

is an axiom and true for all types.

#### 4.6.1.2. Type Inference

If a variable is explicitly declared as type `any`, the inferred type of that variable will always be `any`.

##### 4.6.1.2.1. Default Type of Variables

If a type annotation is missing and no initializer is provided, then the type of a variable is implicitly set to `any`.

In that case, the inferred type of that variable will always be `any` as well. If an initializer is provided, the declared type of the variable will be set to the inferred type of the initializer. Therefore in the latter case, the inferred type of the variable will always be the type of the initializer (cf. [Section 9.1.2, “Variable Statement”](#)).

<sup>5</sup> The rationale for having this limited implementation of type is that API designers already want to start providing hints where later only 32-bit integers will be used. For the time being, **this is checked neither statically nor at runtime!**

If a variable is declared as type `void`, it can be used just as every variable can be used in raw ECMAScript. Since every property can be get and set, the types of properties is inferred as as well. This is formally expressed in [Section 8.1.2, “Identifier”](#).

## 4.6.2. Void Type

The type `void` is used to denote that there is no value at all, as opposed to type `undefined` which denotes that there is a value, but it is always undefined.

The only valid use of type `void` is to declare that a function or method does not return anything. In particular, this means:

- `void` is disallowed as type argument,
- `void` is disallowed as upper/lower bound of type parameters and wild cards,
- when used as return type of functions or methods, `void` may not be nested, i.e.

```
function foo(): void {} // ok
function bar(): any|void {} // error
```

In all the above disallowed cases, type `undefined` should be used instead of `void`.

### 4.6.2.1. Semantics

#### Void Type

##### [Requirement: IDE-13: Void Type \(ver. 1\)](#)

\* The type `void` may only be used as the immediate return type of a function or method.

- If a function  $f$  is declared to return `void`, an error is created if a return statement contains an expression:

$$\begin{aligned} f.returnType = \text{void} \rightarrow \\ \forall r, \mu(r) = \text{ReturnStatement}, r.containingFunction = f : r.expression = \text{null} \end{aligned}$$

- If a function  $f$  is declared to return `void`, an error is issued if the function is called in any statement or expression but an expression statement directly:

$$\begin{aligned} f.returnType = \text{void} \rightarrow \\ \forall e, \text{bind}(e, f) : \mu(e.container) = \text{ExpressionStatement} \end{aligned}$$

The following type hierarchy is defined: `void` is only a subtype of itself but not of any other type and no other type is a subtype of `void`.

`void < : void`

Since `void` cannot be used as the type of variables, fields, formal parameters, etc., a function or method with a return type of `void` cannot be used on the right-hand side of an assignment or in the argument list of a call expression (note the difference to plain JavaScript).

The ECMAScript `void` operator (see [Unary Expressions](#)) has a type of `undefined`, not `void`, because it evaluates to value `undefined` and can be used on the right-hand side of assignments, etc.

## 4.6.3. Unknown Type

Internally N4JS defines the type `unknown`. This type cannot be used by the user. Instead, it is inferred in case of errors. `unknown` behaves almost similar to `any+`. However no error messages once a variable or expression has been inferred to `unknown` in order to avoid consequential errors.

## 4.6.4. Primitive Pathselector and I18nKey

N4JS introduces three new types which are subtypes of string. These types are, in fact, translated to strings and do not add any new functionality. They are solely defined for enabling additional validation.

- `pathSelector<T>` is a generic type for specifying path selector expressions. PathSelectors are used to specify a path to a property in a (JSON-like) model tree.
- The type variable `T` defines the context type (or type of the root of the tree) in which the selector is to be validated. A path selector is defined as a string literal that has to conform to the path selector grammar. The context type is then used to perform a semantic

- `i18nKey` is a string which refers to an internationalization key. The `i18nKey` type is used to reference resource keys specified in resource files. In a project  $p$ , the `i18nKey` type defines the transitive set of all resource keys accessible from  $p$ . Since resource keys are specified as strings, this means that the `i18nKey` type defines a subset of all string literals that can be assigned to a variable of type `i18nKey` in the current project. That means that an assignment of a string literal to a variable of type `i18nKey` is only valid if that string literal is contained in the set defined by `i18nKey`. Resource keys are declared in the properties files of a project and all resource keys from a project are accessible to any project depending on it.

I18nkeys are not yet validated

#### 4.6.4.1. Semantics

The N4JS primitive types `pathSelector<T>`, `i18nKey` and `pathSelector<T>` are basically only marker types of strings for enabling additional validation. Thus, they are completely interchangeable with string types:

```
typeName<T> <: string  string <: typeName<T>
i18nKey <: string  string <: i18nKey
pathSelector<T> <: string  string <: pathSelector<T>
```

As special literals for these N4JS types do not exist, the type has to be explicitly specified in order to enable the additional validation. Note that this validation cannot be applied for more complicated expressions with parts which cannot be evaluated at compile time. For example, `"some.path."segment".prop"` cannot be evaluated at compile time.

### 4.7. Built-in ECMAScript Object Types

N4JS supports all built-in ECMAScript objects [ECMA11a(p.S15)], interpreted as classes. Some of these object types are object versions of primitive types. The object types have the same name as their corresponding primitive type, but start with an upper case letter.

The following types, derived from certain ECMAScript predefined objects and constructs, are supported by means of built-in types as they are required by certain expressions.

- `Object` [ECMA11a(p.p.111)];
- `Function` [ECMA11a(p.p.117)]; representing functions and function objects Section 6.1, “Function Type” but also methods (Section 5.2.3, “Methods”)
- `Array` [ECMA11a(p.p.122)], representing array objects, see Section 4.7.4, “Array Object Type”
- `String` [ECMA11a(p.p.141)]
- `Boolean` [ECMA11a(p.p.141)]
- `Number` [ECMA11a(p.p.141)]
- `RegExp` [ECMA11a(p.p.180)]; they can be constructed by means of special literals (cf. Section 8.1.3, “Literals”)
- global object type
- `Symbol`
- `Promise`
- `Iterator` and `Iterable`

All other ECMAScript types ([ECMA11a(p.S15)], such as `Math`, `Date`, or `Error` are supported by means of predefined classes. ECMAScript 2015 types are defined in the ECMAScript 2015 runtime environment. Since they are defined and used similar to user defined classes, they are not explained in further detail here. These predefined objects are kind of subtypes of `Object`.

#### 4.7.1. Semantics

It is not possible to inherit from any of the built-in ECMAScript object types except for `Object` and `Error`, that is, to use one of these types as supertype of a class. From the N4JS language’s point of view, these built-in types are all final.

#### 4.7.2. Object Type

`Object` [ECMA11a(p.S8.6)] is the (implicit) supertype of all declared (i.e., non-primitive) types, including native types. It models the ECMAScript type `Object`, except that no properties may be dynamically added to it. In order to declare a variable to which properties can be dynamically added, the type `Object+` has to be declared (cf. Section 4.9, “Type Modifiers”).

### 4.7.3. Function-Object-Type

The built-in object type `Function`, a subtype of `Object`, represents all functions, regardless of how they are defined (either via function expression, function declaration, or method declaration). They are described in detail in Section 4.7.3, “Function-Object-Type”.

Since `Function` is the supertype of all functions regardless of number and types of formal parameters, return type, and number and bounds of type parameters, it would not normally be possible to invoke an instance of `Function`. For the time being, however, an instance of `Function` can be invoked, any number of arguments may be provided and the invocation may be parameterized with any number of type arguments (which will be ignored), i.e. [Req-IDE-101] and [Req-IDE-102] do not apply.

### 4.7.4. Array Object Type

The `Array` type is generic with one type parameter, which is the item type. An array is accessed with the index operator, the type of the index parameter is `Number`. The type of the stored values is `typeArgs[0]` (cf. Section 8.1.4, “Array Literal”). Due to type erasure, the item type is not available during runtime, that is to say there are no reflective methods returning the item type of an array.

#### Array Type

**Requirement: IDE-14: Array Type** (ver. 1)

For an array type `A`, the following conditions must be true:

- $|A.typeArgs| = 1$

### 4.7.5. String Object Type

Object type version of `string`. It is highly recommend to use the primitive version only. Note that is is not possible to assign a primitive typed value to an object typed variable.

### 4.7.6. Boolean Object Type

Object type version of `boolean`. It is highly recommend to use the primitive version only. Note that is is not possible to assign a primitive typed value to an object typed variable.

### 4.7.7. Number Object Type

Object type version of `number`. It is highly recommend to use the primitive version only. Note that is is not possible to assign a primitive typed value to an object typed variable.

### 4.7.8. Global Object Type

This is the globally accessible namespace which contains element such as `undefined`, and in case of browsers, `window`. Depending on the runtime environment, the global object may has different properties defined by means of dynamic polyfills.

### 4.7.9. Symbol

The symbol constructor function of ECMAScript 2015. Support for symbols is kept to a minimum in N4JS:

- creating symbols with `var sym = Symbol("description")` is not supported.
- creating shared symbols with `var sym = Symbol.for("key")` is not supported. Also the inverse `Symbol.keyFor(sym)` is not supported.
- retrieving built-in symbols via properties in `Symbol` is supported, however, the only built-in symbol available in N4JS is the iterator symbol that can be retrieved with `Symbol.iterator`.

The rationale for this selective support for symbols in N4JS is to allow for the use (and custom definition) of iterators and iterables and their application in the `for...of` loop with as little support for symbols as possible.

### 4.7.10. Promise

`Promise` is provided as a built-in type as in ECMAScript 2015. Also see Section 6.4.1, “Asynchronous Functions” for asynchronous functions.

### 4.7.11. Iterator Interface

A structurally typed interface for *iterators* as defined by the ECMAScript 6 iterator protocol.

#### Iterable in N4JS

```
// providedByRuntime
export public interface ~Iterator<T> {
    public next(): IteratorEntry<T>
}

// providedByRuntime
export public interface ~IteratorEntry<T> {
    public done: boolean;
    public value: T?;
}
```

Interface `IteratorEntry` was introduced mainly as a workaround; this interface could be removed and replaced with a corresponding structural type reference as return type of method `next()`.

### 4.7.12. Iterable Interface

A structurally typed interface for objects that can be iterated over, i.e. *iterables* as defined by the ECMAScript 6 iterator protocol.

```
// providedByRuntime
export public interface ~Iterable<T> {
    public [Symbol.iterator](): Iterator<T>
}
```

Note that this interface's method is special in that a symbol is used as identifier. You can use the ordinary syntax for computed property names in ECMAScript 6 for overriding / implementing or invoking this method.

## 4.8. Built-In N4JS Types

N4JS additionally provides some built-in classes which are always available with the need to explicitly import them.

### 4.8.1. N4Object

Although `N4Object` is a built-in type, it is not the default supertype. It is a subtype of `Object`.

#### 4.8.1.1. Semantics

`N4Object < : Object`

### 4.8.2. N4Class

The type `N4Class` is used for extended reflection in N4JS.

Add further docs for this type

### 4.8.3. IterableN

Work in progress.

Currently there are built-in types `Iterable2<T1, T2>` ... `Iterable9<T1, ..., T9>`. They are mainly intended for type system support of array destructuring literals.



This is not documented in detail yet, because we want to gain experience with the current solution first, major refinement may be incoming.

## 4.9. Type Modifiers

Type expressions can be further described with type modifiers. The type modifiers add additional constraints to the type expression which are then used to perform a stricter validation of the source code. Type modifiers can not be used in type arguments.

The general type modifiers *nullable*, *nonnull* and *dynamic* can be used for variables, attributes, method parameters and method types. Optional and variadic modifiers can only be applied for formal parameters.

### 4.9.1. Dynamic

The dynamic type modifier marks a type as being dynamic. A dynamic type behaves like a normal JavaScript object, so you can read/write any property and call any method on it. The default behavior for a type is to be static, that is no new properties can be added and no unknown properties can be accessed.

`T < : T+ and T+ < : T` is always true. Using dynamically added members of a dynamic type is never type safe. Using the `delete` operator on a subtype of `NativeObject` is not allowed.

#### Non-Dynamic Primitive Types

##### Requirement: IDE-15: Non-Dynamic Primitive Types (ver. 1)

1. All primitive types except `any` must not be declared dynamic.
2. Only parameterized type references and this type reference can be declared dynamic.<sup>6</sup>

### 4.9.2. Optional Return Types

Only formal parameters and return types can be marked as optional.

An optional return type indicates that the function / method need not be left via a return statement with an expression; in that case the return value is `undefined`. For constraints on using the optional modifier, see [Section 4.7.3, “Function-Object-Type”](#).

## 4.10. Union and Intersection Type (Composed Types)

Given two or more existing types, it is possible to compose a new type by forming either the union or intersection of the base types. The following example gives a small overview of common use cases of union and intersection types.

#### Example 4.12. Composed types

This example shows how union and intersection types affect the types of their field members in case the fields have different types. It is for illustration purposes only. The type of the composed field depends on the access type: When reading, the field type of an intersection/union also resolves to the intersection/union. In contrast, when writing a field, the field type of an intersection/union resolves to the union/intersection respectively.

```
interface A { f : int = 1; }
interface B { f : string = "h"; }

class CA implements A {}
class CB implements B {}

let aub : A|B; // union type
let aib : A&B; // intersection type

function u() {
    aub = (catIsAlive)? new CA() : new CB(); // catIsAlive is boolean
    let x = aub.f; // x = {1 | "h"}
    aub.f = undefined; // undefined can be assigned to int and string types
}
function i() {
    let a = aib as A;
    let b = aib as B;
    a.f = 23;
    b.f = "text";
    let x = aib.f; // x = {23 & "text"} which is impossible
}
// type of 'aub.f' --> int|string
let fu = aub.f;
// type of 'aub.f' --> int&string
aub.f = undefined;
// type of 'aib.f' --> int&string
let fi = aib.f;
// type of 'aib.f' --> int|string
aib.f = undefined;
```

<sup>6</sup> This is a consequence of the syntax definition.

Note that no instance `aib` of intersection type `A&B` can be instantiated, since the instance's class would have to define a field `f` which would have to comply to both of the interfaces `A` and `B`. Still the function `i()` shows in general how variables of intersection types can be casted and accessed.

The following sections define these *union* and *intersection types* in detail.

## 4.10.1. Union Type

Union type reflect the dynamic nature of JavaScript. Union types can be used almost everywhere (e.g., in variable declarations or in formal method parameters). The type inferencer usually avoids returning union types and prefers single typed joins or meets. *The most common use case for union types is for emulating method overloading*, as we describe later on.<sup>7</sup>

### 4.10.1.1. Syntax

For convenience, we repeat the definition of union type expression:

```
UnionTypeExpression: 'union' '(' typeRefs+=TypeRefWithoutModifiers (',' typeRefs+=TypeRefWithoutModifiers)* ')' ;
```

### 4.10.1.2. Semantics

An union type states that the type of a variable may be one or more types contained in the union type. In other words, a union type is a kind of type set, and the type of a variable is contained in the type set. Due to interfaces, a variable may conform to multiple types.

#### Union Type

##### Requirement: IDE-18: Union Type (ver. 1)

For a given union type  $U = \text{union}\{T_1, \dots, T_n\}$ , the following conditions must hold:

1. Non-empty: At least one element has to be specified:  
 $U.typeRefs \neq \emptyset (n \geq 1)$
2. Non-dynamic: The union type itself must not be declared dynamic:  
 $\neg U.dynamic$
3. Non-optional elements:  
 $\forall T \in U.typeRefs \rightarrow \neg T.opt$

#### Union Type Subtyping Rules

##### Requirement: IDE-19: Union Type Subtyping Rules (ver. 1)

Let  $U$  be an union type.

- The union type is a common supertype of all its element types:

$$\frac{T \in U.typeRefs}{T < : U}$$

- More generally, a type is a subtype of a union type, if it is a subtype of at least one type contained in the union:

$$\frac{\exists T \in U.typeRefs: S < : T}{S < : U}$$

- A union type is a subtype of a type  $S$ , if all types of the union are subtypes of that type. This rule is a generalization of the sub typing rules given in [Igarashi07a(p.p.40)]

$$\frac{\forall T \in U.typeRefs: T < : S}{U < : S}$$

- Commutativity: The order of element does not matter:

$$\text{union}\{A, B\} = \text{union}\{B, A\}$$

- Associativity:  $\text{union}\{A, \text{union}\{B, C\}\} = \text{union}\{\text{union}\{A, B\}, C\}$

- Uniqueness of elements: A union type may not contain duplicates (similar to sets):

<sup>7</sup> For type theory about union types, [Pierce02a(p.15.7)] and [Igarashi07a], other languages that explicitly support the notion of union type include Ceylon [King13a(p.3.2.4/5)]

$$\forall 1 \leq i < k \leq n, \text{union}\{T_1, \dots, T_n\} : T_i \neq T_k$$

### Implicit simplification of union types

**Requirement: IDE-20:** Implicit simplification of union types (ver. 1)

Let  $U$  be an union type. The following simplification rules are always automatically applied to union types.

- Simplification of union type with one element: If a union type contains only one element, it is reduced to the element:

$$\frac{\text{union}\{T\}}{T}$$

- Simplification of union types of union types: A union type  $U$  containing another union type  $V$  is reduced to a single union type  $W$ , with  $W.\text{typeRefs} = U.\text{typeRefs} \cup V.\text{typeRefs}$ :

$$\frac{\text{union}\{S_1, \dots, S_{k-1}, \text{union}\{T_1, \dots, T_m\}, S_{k+1}, \dots, S_n\}}{\text{union}\{S_1, \dots, S_{k-1}, T_1, \dots, T_m, S_{k+1}, \dots, S_n\}}$$

- Simplification of union type with undefined or null: Since undefined is the bottom type, and null is kind of a second bottom type, they are removed from the union:

$$\frac{\text{union}\{T_1, \dots, T_{k-1}, T_k, \dots, T_n\}}{\text{union}\{T_1, \dots, T_{k-1}, \text{undefined}\}, T_k, \dots, T_n}$$

$$\frac{\text{union}\{T_1, \dots, T_{k-1}, T_k, \dots, T_n\}}{\text{union}\{T_1, \dots, T_{k-1}, \text{null}\}, T_k, \dots, T_n}$$



Simplification rules for union types with one element are applied first.

- The structural typing strategy is propagated to the types of the union:

$$\frac{\sim \text{union}\{T_1, \dots, T_n\}}{\text{union}\{\sim T_1, \dots, \sim T_n\}}$$

Remarks:

- The simplification rules may be applied recursively.
- For given types  $B < : A$ , and the union type  $U = \text{union}\{A, B\}$ ,  $U \neq B$ . The types are equivalent, however:  $A < : = U$  and  $U < : = A$ .<sup>8</sup>

#### Example 4.13. Subtyping with union type

Let  $A$ ,  $B$ , and  $C$  be defined as in the chapter beginning ( $C < : B < : A$ )

The following subtyping relations with union types are to be evaluated as follows:<sup>9</sup>

$A < : \text{union}\{A\}$	$\rightarrow \text{true}$
$A < : \text{union}\{A, B\}$	$\rightarrow \text{true}$
$B < : \text{union}\{A, B\}$	$\rightarrow \text{true}$
$C < : \text{union}\{A, B\}$	$\rightarrow \text{true}$
$A < : \text{union}\{B, C\}$	$\rightarrow \text{false}$
$B < : \text{union}\{B, C\}$	$\rightarrow \text{true}$
$C < : \text{union}\{B, C\}$	$\rightarrow \text{true}$
$\text{union}\{A\} < : A$	$\rightarrow \text{true}$
$\text{union}\{B\} < : A$	$\rightarrow \text{true}$
$\text{union}\{B, C\} < : A$	$\rightarrow \text{true}$
$\text{union}\{A, B\} < : B$	$\rightarrow \text{false}$
$\text{union}\{X, Z\} < : \text{union}\{Z, X\}$	$\rightarrow \text{true}$
$\text{union}\{X, Y\} < : \text{union}\{X, Y, Z\}$	$\rightarrow \text{true}$
$\text{union}\{X, Y, Z\} < : \text{union}\{X, Y\}$	$\rightarrow \text{false}$

The simplification constraints are used by the type inferrer. It may be useful, however, to define union types with superfluous elements, as the next example demonstrates

<sup>8</sup> This is different from Ceylon ([King13a(p.3.2.3)]), in which the union is defined to be `the same type as`  $A$ . Although the meaning of `same` is not clear, it is possibly used as a synonym for `equivalent`.

<sup>9</sup> See Example Example 4.1, "Type Examples, Class Hierarchy" for class definitions.

**Example 4.14. Superfluous elements in union type**

```
class A{}
class B extends A{}
class C extends A{}

function foo(p: union{A,B}) {...}
```

Although `B` is superfluous, it may indicate that the function handles parameters of type differently than one of type `A` or `C`.

Although a union type is a [LCST](#) of its contained (non-superfluous) types, the type inferrer usually does not create new union types when computing the join of types. If the join of types including at least one union type is calculated, the union type is preserved if possible. The same is true for meet.

For the definition of join and meet for union types, we define how a type is added to a union type:

**Union of union type****Requirement: IDE-21: Union of union type (ver. 1)**

The union of union types is defined similar to the union of sets. The union is not simplified, but it contains no duplicates.

If a type `A` is contained in a union type, then the union type is a common supertype, and (since it is the union itself) also the [LCST](#) of both types. This finding is the foundation of the definition of join of a (non-union) type with a union type:

**Join with Union Type****Requirement: IDE-22: Join with Union Type (ver. 1)**

The join  $J$  of a union type  $U$  with a type  $T$  is the union of both types:

$$\frac{J = U \cup T}{(U \vee T) = J}$$

Remarks:

- Joining a union type with another type is not similar to joining the elements of the union type directly with another type. That is
- $$A \text{ join } \text{union}\{B, C\} \neq A \text{ join } B \text{ join } C$$
- The computed join is simplified according to the constraints defined above.

**Meet with Union Type****Requirement: IDE-23: Meet with Union Type (ver. 1)**

The meet of union types is defined as the meet of the elements. That is

$$\frac{\begin{array}{c} T_1 \wedge S_1 \dots \wedge T_n \wedge S \\ \hline \text{union}\{T_1 \dots T_n\} \wedge S \end{array}}{\begin{array}{c} T_1 \wedge S_1 \dots T_1 \wedge S_m \dots T_n \wedge S_1 \dots T_n \wedge S_m \\ \hline \text{union}\{T_1 \dots T_n\} \wedge \text{union}\{S_1 \dots S_m\} \end{array}}$$

Remarks:

- The meet of a union type with another type is not a union type itself. This gets clear when looking at the definition of meet and union type. While for a given  $U = \text{union}\{A, B\}$ ,  $A < : U$  and  $B < : U$ , the opposite  $U < : A$  is usually not true (unless  $U$  can be simplified to  $A$ ). So, for  $A \wedge U$ , usually  $U$  cannot be the meet.

**Upper and Lower Bound of a Union Type****Requirement: IDE-24: Upper and Lower Bound of a Union Type (ver. 1)**

The upper and lower bound of a union type  $U$  is a union type  $U'$  containing the upper and lower bound of the elements of  $U$ :

```
upper(\text{union}\{T_1, \dots, T_n\}) := \text{union}\{\text{upper}(T_1), \dots, \text{upper}(T_n)\}
lower(\text{union}\{T_1, \dots, T_n\}) := \text{union}\{\text{lower}(T_1), \dots, \text{lower}(T_n)\}
```

#### 4.10.1.3. Warnings

In case the `any` type is used in a union type, all other types in the union type definition become obsolete. However, defining other types along with the `any` type might seem reasonable in case those other types are treated specifically and thus are mentioned explicitly in the definition. Nevertheless the use of the `any` type produces a warning, since its use can indicate a misunderstanding of the union type concept and since documentation can also be done in a comment.

Any type in union types

**Requirement: IDE-25: Any type in union types (ver. 1)**

No union type shall contain an type:

$\nexists any \in U.typeRefs$

Similar to the documentary purpose of using specific classes along with the `any` type is the following case. When two types are used, one of them a subtype of the other, then this subtype is obsolete. Still it can be used for documentary purposes. However, a warning will be produced to indicate unnecessary code. The warning is only produced when both of the types are either classes or interfaces, since e.g. structural types are supertypes of any classes or interfaces.

Redundant subtypes in union types

**Requirement: IDE-26: Redundant subtypes in union types (ver. 1)**

Union types shall not contain class or interface types which are a subtype of another class or interface type that also is contained in the union type.

$\nexists TT \in U.typeRefs \exists T \in U.typeRefs : (TT < : T \wedge isClassOrInterface(T) \wedge isClassOrInterface(TT))$

### 4.10.2. Intersection Type

Intersection type reflects the dynamic nature of JavaScript, similar to union type. As in Java, intersection type is used to define the type boundaries of type variables in type parameter definitions. They are inferred by the type inferencer for type checking (as a result of join or meet). In contrast to Java, however, intersection type can be declared explicitly by means of intersection type expression.<sup>10</sup>

#### 4.10.2.1. Syntax

For convenience, we repeat the definition of intersection type expression and of type variables in which intersection types can be defined as in Java:

```
IntersectionTypeExpression: 'intersection' '(' typeRefs+=TypeRefWithoutModifiers (',' typeRefs+=TypeRefWithoutModifiers)* ')';
TypeVariable: name=IDENTIFIER ('extends' declaredUpperBounds+=ParameterizedTypeRefNominal ('&' declaredUpperBounds+=ParameterizedTypeRefNominal)*)?
```

#### 4.10.2.2. Semantics

An intersection type may contain several interfaces but only one class. It virtually declares a subclass of this one class and implements all interfaces declared in the intersection type. If no class is declared in the intersection type, the intersection type virtually declares a subclass of an N4Object instead. This virtual subclass also explains why only one single class may be contained in the intersection.

Intersection Type

**Requirement: IDE-27: Intersection Type (ver. 1)**

For a given intersection type  $I$ , the following conditions must hold:

1. The intersection must contain at least one type:

$I.typeRefs \neq \emptyset$

2. Only one class must be contained in the intersection type:

<sup>10</sup> For type theory about intersection types, see [Pierce02a(p.15.7)] and [Laurent12a], other languages supporting explicit notion of intersection type include Ceylon [King13a(p.3.2.4/5)].

$$(\exists C \in I.typeRefs : \mu(C) = \text{Class}) \rightarrow \# T \in I.typeRefs \setminus \{C\} : \mu(T) = \text{Class}$$

For the time being, only a warning is produced when more than one class is contained in the intersection type .

### 3. Non-optional elements:

$$\forall T \in I.typeRefs \rightarrow \neg T.opt$$

## Intersection Type Subtyping Rules

### Requirement: IDE-175: Intersection Type Subtyping Rules (ver. 1)

Let  $I$  be an intersection type.

- An intersection type is a subtype of another type, if at least one of its contained types is a subtype of that type: <sup>11</sup>

$$\frac{\exists T \in I.typeRefs : T < : S}{I < : S}$$

- A type is a subtype of an intersection type, if it is a subtype of all types contained in the intersection type: <sup>12</sup>

$$\frac{\forall T \in I.typeRefs : S < : T}{S < : I}$$

- Non-optional elements:  $\forall T \in I.typeRefs \rightarrow \neg T.opt$

## Implicit simplification of intersection types

### Requirement: IDE-28: Implicit simplification of intersection types (ver. 1)

Let  $I$  be an intersection type. The following simplification rules are always automatically applied to intersection types.

- The structural typing strategy is propagated to the types of the intersection:

$$\frac{\sim \text{intersection}\{T_1, \dots, T_n\}}{\text{intersection}\{\sim T_1, \dots, \sim T_n\}}$$

These subtyping rules are similar to Ceylon. <sup>13</sup>

During validation, intersection types containing union or other intersection types may be inferred. In this case, the composed types are flattened. The aforementioned constraints must hold. We also implicitly use this representation in this specification.

### Example 4.15. Subtyping with intersection type

Let A, B, and C be defined as in the chapter beginning ( $C < : B < : A$ )

The following subtyping relations with intersection types are to be evaluated as follows: <sup>14</sup>

A <: intersection{A}	-> true
A <: intersection{A, A}	-> true
intersection[A, X] <: A	-> true
intersection[X, A] <: A	-> true
A <: intersection{A, X}	-> false
intersection[A, X] <: intersection[X, A]	-> true
H12 <: intersection[I1, I2]	-> true
intersection[I1, I2] <: H12	-> false
H1 <: intersection[I1, I2]	-> false
H23 <: intersection[I1, I2]	-> false
B <: intersection{A}	-> true
intersection[I1, I2] <: I	-> true
H12 <: intersection[I, I2]	-> true
A <: intersection{A, Any}	-> true
intersection[A, Any] <: A	-> true

The join of intersection types is defined as the join of the elements. That is:

$$\frac{T_1 \vee S \vee \dots \vee T_n \vee S}{\text{intersection}\{T_1, \dots, T_n\} \vee S}$$

<sup>11</sup> This rule is a generalization of the subtyping rules given in [Laurent12a] Table 2,  $\cap_l^1$  and  $\cap_l^2$

<sup>12</sup> This rule is a generalization of the subtyping rules given in [Laurent12a] Table 2,  $\cap_r$

<sup>13</sup> In Ceylon, for a given union type  $U = T_1 \parallel T_2$  and intersection type  $I = T_1 \& T_2$  (with ' $\parallel$ ' is union and ' $\&$ ' is intersection),  $T_1 < : U$  and  $T_2 < : U$  is true, and  $T_1 < : I$  and  $T_2 < : I$  is true. We should define that as well (if it is not already defined). Cf [King13a(p.3.2.4/5)]

<sup>14</sup> See Example Example 4.1, "Type Examples, Class Hierarchy" for class definitions.

$$\frac{T_1 \vee S_1, \dots, T_1 \vee S_m, \dots, T_n \vee S_1, \dots, T_n \vee S_m}{\text{intersection}\{T_1, \dots, T_n\} \vee \text{intersection}\{S_1, \dots, S_m\}}$$

The meet of intersection types is defined over their elements. That is:

$$\frac{\begin{array}{c} \text{intersection}\{T_1 \wedge S, \dots, T_n \wedge S\} \\ \text{intersection}\{T_1, \dots, T_n\} \wedge S \end{array}}{\text{intersection}\{T_1 \wedge S_1, \dots, T_1 \wedge S_m, \dots, T_n \wedge S_1, \dots, T_n \wedge S_m\}} \\ \frac{}{\text{intersection}\{T_1, \dots, T_n\} \wedge \text{intersection}\{S_1, \dots, S_m\}}$$

The upper and lower bound of an intersection type  $I$  is a union type  $I'$  containing the upper and lower bound of the elements of  $I$ :

$$\begin{array}{l} \text{upper}(\text{intersection}\{T_1, \dots, T_n\}) := \text{intersection}\{\text{upper}(T_1), \dots, \text{upper}(T_n)\} \\ \text{lower}(\text{intersection}\{T_1, \dots, T_n\}) := \text{intersection}\{\text{lower}(T_1), \dots, \text{lower}(T_n)\} \end{array}$$

#### 4.10.2.3. Warnings

Using `any` types in intersection types is obsolete since they do not change the resulting intersection type. E.g. the intersection type of A, B and `any` is equivalent to the intersection type of A and B. However, using the `any` type is no error because it can be seen as a neutral argument to the intersection. Nevertheless the use of the `any` type produces a warning, since its use can indicate a misunderstanding of the intersection type concept and since it always can be omitted.

Any type in intersection types

**Requirement: IDE-32: Any type in intersection types (ver. 1)**

No intersection type shall contain an type:

$\nexists \text{any} \in I.\text{typeRefs}$

The use of the `any` type in an intersection type is similar to the following case. When two types are used, one of them a supertype of the other, then this supertype is obsolete. Hence, a warning will be produced to indicate unnecessary code. The warning is only produced when both of the types are either classes or interfaces, since e.g. structural types are supertypes of any classes or interfaces.

Redundant supertypes in intersection types

**Requirement: IDE-33: Redundant supertypes in intersection types (ver. 1)**

Intersection types shall not contain class or interface types which are a supertype of another class or interface type that also is contained in the intersection type.

$\nexists T \in I.\text{typeRefs} : \exists TT \in I.\text{typeRefs} : (TT < : T \wedge \text{isClassOrInterface}(T) \wedge \text{isClassOrInterface}(TT))$

#### 4.10.3. Composed Types in Wildcards

Composed types may appear as the bound of a wildcard. The following constraints apply: <sup>15</sup>

Composed Types as Bound of a Wildcard

**Requirement: IDE-34: Composed Types as Bound of a Wildcard (ver. 1)**

A composed type may appear as the upper or lower bound of a wildcard. In the covariant case, the following subtype relations apply:

```
union{ G<? extends A>, G<? extends B> } \subtype G<? extends union[A,B]>
G<? extends intersection[A,B]> \subtype intersection{ G<? extends A>, G<? extends B> }
```

In the contra variant case, the following subtype relations apply:

```
union{ G<? super A>, G<? super B> } \subtype G<? super intersection[A,B]>
G<? super union[A,B]> \subtype intersection{ G<? super A>, G<? super B> }
```

<sup>15</sup> see "Covariance and contravariance with unions and intersections" at <http://ceylon-lang.org/documentation/1.1/tour/generics/>

#### 4.10.4. Property Access for Composed Types

It is possible to directly access properties of union and intersection types. The following sections define which properties are accessible.

##### 4.10.4.1. Properties of Union Type

As an (oversimplified) rule of thumb, the properties of a union type  $U = T_1 \sqcup T_2$  are simply the intersection of the properties  $U.properties = T_1.properties \cap T_2.properties$ . In other words, a property 'p' in the union type is the least common denominator of all 'p' in  $T_{\{1\}}$  and  $T_{\{2\}}$ . It is not quite that simple, however, as the question of "equality" with regards to properties has to be answered.

For a given union type  $U = T_1 \sqcup T_2$ , the following constraints for its members must hold:

$\forall a \in U.attributes:$

$$\begin{aligned} \forall k \in \{1, 2\} : \exists a_k \in T_k.attributes : & a_k.acc > private \\ & \wedge a.acc = \min(a_1.acc, a_2.acc) \\ & \wedge a.name = a_1.name = a_2.name \\ & \wedge a.typeRef = a_1.typeRef = a_2.typeRef \end{aligned}$$

$\forall m \in U.methods:$

$\exists m_1 \in T_1.methods, m_2 \in T_2.methods,$   
**with**  $p = m.fpars \wedge p' = m_1.fpars \wedge p'' = m_2.fpars$ , **WLOG**  $|p| \leq |p'|$ :

$$\begin{aligned} & \forall k \in \{1, 2\} : m_k.acc > private \\ & \wedge m.acc = \min(m_1.acc, m_2.acc) \\ & \wedge m.name = m_1.name = m_2.name \\ & \wedge m.typeRef = m_1.typeRef \mid m_2.typeRef \\ & \wedge \forall i < |p| : p_i \text{ exists with} \\ p_i.name &= \begin{cases} p''_i.name & i \geq |p| \vee p'_i.name = p''_i.name \\ p'_i.name + "-" + p''_i.name & \text{else} \end{cases} \\ p_i.typeRef &= \begin{cases} p'_i.typeRef \& p''_i.typeRef & i < |p| \\ p'_{|p|+1}.typeRef \& p''_{|p|+1}.typeRef & i \geq |p| \wedge p'_i \neq p''_i.var \\ p''_i.typeRef & \text{else} \end{cases} \\ p_i.opt &= \begin{cases} (p'_i.opt \wedge p''_i.opt) & i < |p| \\ p''_i.opt & \text{else} \end{cases} \\ p_i.var &= \begin{cases} p'_i.var \wedge p''_i.var & i < |p| \wedge i = |p| - 1 \\ p''_i.var & i \geq |p| \wedge i = |p| - 1 \\ false & \text{else} \end{cases} \\ & \wedge (l = |p| = |p'| \wedge \neg (p'_l \neq l.opt \wedge p''_l \neq l.opt)) \wedge \exists v \in \{p'^l-1, p''^l-1\}.v.var : p_l \text{ exists with} \\ p_l.name &= v.name \\ p_l.typeRef &= v.typeRef \\ p_l.opt &= true \\ p_l.var &= true \end{aligned}$$

The following table shows how non-method members of union types are merged. The resulting member type depends on whether the member is being accessed during a read (R) or write (W) operation. The type of a field, of a getter or of the parameter of a setter is indicated in brackets.

Table 4.1. Merged Members of Unions

Members	S=T		S#T	
	R	W	R	W
<b>field:S   field:T</b>	field:S		getter:S T	setter:S&T
<b>getter:S   getter:T</b>	getter:S	-	getter:S T	-
<b>setter:S   setter:T</b>	-	setter:S	-	setter:S&T
<b>field:S   getter:T</b>	getter:S	-	getter:S T	-
<b>field:S   setter:T</b>	-	setter:S	-	setter:S&T
<b>getter:S   setter:T</b>	-	-	-	-

#### 4.10.4.1.1. Remarks on union type's members:

- Fields of the same type are merged to a composed field with the same type. Fields of different types are merged to a getter and setter.
- The return type of a composed getter is the *union* type of the return types of the merged getters.
- The type of a composed setter is the *intersection* type of the types of the merged setters.
- Fields can be combined with getters and/or setters:
  - fields combined with getters allow read-access.
  - non-const fields combined with setters allow write-access.
  - non-const fields combined with getters *and* setters, i.e. each type has either a non-const field or both a getter and a setter of the given name, allow both read- and write-access.

Again, types need not be identical; for read-access the *union* of the fields' types and the getters' return types is formed, for write-access the *intersection* of the fields' types and the setters' types is formed. In the third case above, types are combined independently for read- and write-access if the getters and setters have different types.

- The name of a method's parameter is only used for error or warning messages and cannot be referenced otherwise.
- The return type of a composed method is the *union* type of the return types of the merged methods.
- A composed method parameter's type is the *intersection* type of the merged parameters types.

#### 4.10.4.2. Properties of Intersection Type

As an (oversimplified) rule of thumb, the properties of an intersection type  $I = T_1 \& T_2$  are the union of properties  $I.properties = T_1.properties \cup T_2.properties$ . In other words, a property 'p' in the union type is the greatest common denominator of all 'p' in  $T_{\{1\}}$  and  $T_{\{2\}}$ . It is not quite that simple, however, as the question of "equality" with regards to properties has to be answered.

##### Members of an Intersection Type

**Requirement: IDE-36: Members of an Intersection Type (ver. 1)**

For a given intersection type  $I = T_1 \& T_2$ , the following constraints for its members must hold:

$\forall a \in I.attributes:$

$$\begin{aligned}
 & (\exists a_1 \in T_1.attributes, a_1.acc > private) \vee (\exists a_2 \in T_2.attributes, a_2.acc > private) \\
 & \wedge a.name = \begin{cases} a_1.name & a_1 \neq null \wedge (a_2 = null \vee a_2.name = a_1.name) \\ a_2.name & \text{else} \end{cases} \\
 & \wedge a.acc = \begin{cases} a_1.acc & a_1 \neq null \wedge (a_2 = null \vee a_2.acc \leq a_1.acc) \\ a_2.acc & \text{else} \end{cases} \\
 & \wedge a.typeRef = \begin{cases} a_1.typeRef \& a_2.typeRef & a_1 \neq null \wedge a_2 \neq null \\ a_1.typeRef & & a_1 \neq null \\ a_2.typeRef & & \text{else } (a_2 \neq null) \end{cases}
 \end{aligned}$$

$\forall m \in I.methods:$

$(\exists m_1 \in T_1.methods, m_1.acc > private) \vee (\exists m_2 \in T_2.methods, m_2.acc > private);$

$$\begin{aligned}
 & \text{with } p = mfpars \\
 & \wedge \text{if } m_1 \text{ exists } p' = m_1fpars \text{ (else } p' = \emptyset\text{),} \\
 & \wedge \text{if } m_2 \text{ exists } p'' = m_2fpars \text{ (else } p'' = \emptyset\text{), WLOG } |p'| \leq |p''|: \\
 m.name &= \begin{cases} m_1.name & m_1 \neq null \wedge (m_2 = null \vee m_2.name = m_1.name) \\ m_2.name & \text{else} \end{cases} \\
 \wedge m.acc &= \begin{cases} m_1.acc & m_1 \neq null \wedge (m_2 = null \vee m_2.acc \leq m_1.acc) \\ m_2.acc & \text{else} \end{cases} \\
 \wedge m.typeRef &= \begin{cases} m_1.typeRef \& m_2.typeRef & m_1 \neq null \wedge m_2 \neq null \\ m_1.typeRef & m_1 \neq null \\ m_2.typeRef & \text{else } (m_2 \neq null) \end{cases} \\
 \wedge \forall i < |p'|: p_i \text{ exists with} \\
 p_i.name &= \begin{cases} p''_i.name & i \geq |p'| \vee p''_i.name = p'_i.name \\ p'_i.name + "-" + p''_i.name & \text{else} \end{cases} \\
 p_i.typeRef &= \begin{cases} p'_i.typeRef \mid p''_i.typeRef & i < |p'| \\ p''_{|p'|+1}.typeRef \mid p''_i.typeRef & i \geq |p'| \wedge p''_{|p'|+1}.var \\ p''_i.typeRef & \text{else} \end{cases} \\
 p_i.opt &= (\exists k \leq \min(|p'| - 1, i) : p'^k.opt) \vee (\exists k \leq i : p''^k.opt) \\
 p_i.var &= \begin{cases} p'_i.opt \vee (p''_i.var \vee p''_i.var) & i < |p'| \wedge i = |p'| - 1 \\ p''_i.var & i \geq |p'| \wedge i = |p'| - 1 \\ false & \text{else} \end{cases} \\
 \wedge (l = |p'| = |p''| \wedge l > 0 \wedge \neg (p_{l-1}.opt) \wedge \exists v \in \{p'^{l-1}, p''^{l-1}\}.var : p_l \text{ exists with} \\
 p_l.name &= v.name \\
 p_l.typeRef &= v.typeRef \\
 p_l.opt &= true \\
 p_l.var &= true
 \end{aligned}$$

The following table shows how non-method members of intersection types are merged. The resulting member type depends on whether the member is being accessed during a read (R) or write (W) operation. The type of a field, of a getter or of the parameter of a setter is indicated in brackets.

Table 4.2. Merged Members of Intersections

Members	S=T		S#T	
	R	W	R	W
field:S & field:T	field:S		getter:S&T	setter:S T
getter:S & getter:T	getter:S	-	getter:S&T	-
setter:S & setter:T	-	setter:S	-	setter:S T
field:S & getter:T	field:S		getter:S&T	setter:S
field:S & setter:T	field:S		getter:S	setter:S T
getter:S & setter:T	field:S		getter:S	setter:T

#### 4.10.4.2.1. Remarks on intersection type's methods:

- The name of a method's parameter is only used for error or warning messages and cannot be referenced otherwise.

- The return type of a method is the *intersection* type of the return types of the merged methods.
- A method parameter's type is the *union* type of the merged parameters types.

## 4.11. Constructor and Classifier Type

A class definition as described in [Section 5.1.3, “Classes”](#) declares types. Often, it is necessary to access these types directly, for example to access staticmembers or for dynamic construction of instances. These two use cases are actually slightly different and N4JS provides two different types, one for each use case: constructor and classifier type.<sup>16</sup> The constructor is basically the classifier type with the additional possibility to call it via `new` in order to create new instances of the declared type.

Both `meta` types are different from Java's type `Class<T>`, as the latter has a defined set of members, while the N4JS metatypes will have members according to a class definition. The concept of constructors as metatypes is similar to ECMAScript 2015 [[ECMA15a\(p.14.5\)](#)].

### 4.11.1. Syntax

```
ConstructorTypeRef returns ConstructorTypeRef: 'constructor' '{' typeArg = [TypeArgument] '}';
ClassifierTypeRef returns ClassifierTypeRef: 'type' '{' typeArg = [TypeRef] '}';
```

### 4.11.2. Semantics

1. Static members of a type  $T$  are actually members of the classifier type `type{T}`.
2. The keyword `this` in a static method of a type  $T$  actually binds to the classifier type `type{T}`.
3. The constructor type `constructor{T}` is a subtype of the classifier type `type{T}`:

$$\forall T: \text{constructor}\{T\} < : \text{type}\{T\}$$

4. If a class  $B$  is a subtype (subclass) of a class  $A$ , then the classifier type `type{B}` also is a subtype of `type{A}`:

$$\frac{B < : A}{\text{type}\{B\} < : \text{type}\{A\}}$$

5. If a class  $B$  is a subtype (subclass) of a class  $A$ , and if the constructor function of  $B$  is a subtype of the constructor function of  $A$ , then the classifier type `constructor{B}` also is a subtype of `constructor{A}`:

$$\frac{B < : A \quad \text{Bctor} < : \text{Actor}}{\text{constructor}\{B\} < : \text{constructor}\{A\}}$$

The subtype relation of the constructor function is defined in [Section 6.1, “Function Type”](#). In the case of the default `N4Object` constructor, the type of the object literal argument depends on required attributes.

This subtype relation for the constructor type is enforced if the constructor of the super class is marked as `final`, see [Section 4.11, “Constructor and Classifier Type”](#) for details.

6. The type of a classifier declaration or classifier expression is the constructor of that class:

$$\frac{\mu(C) \in \{\text{classifierDefinition}\}}{\Gamma \vdash C: \text{constructor}[C]}$$

7. A class cannot be called as a function in ECMAScript. Thus, the constructor and type type are only subtype of `Object`:

$$\begin{aligned} \forall T: \\ \text{constructor}\{T\} < : \text{Object} \\ \text{type}\{T\} < : \text{Object} \end{aligned}$$

8. If the type argument of the constructor is not a declared type (i.e., a wildcard or a type variable with bounds), the constructor cannot be used in a new expression. Thus, the constructor function signature becomes irrelevant for subtype checking. In that case, the following rules apply:

<sup>16</sup> The classifier type is, in fact, the `type type` or `metatype` of a type. We use the term classifier type in the specification to avoid the bogus `type type` terminology.

$$\frac{\text{Supper} < : T.\text{upper} \quad T.\text{lower} < : S.\text{lower} \quad \mu(T) \neq \text{DeclaredTypeWithAccessModifier}}{\text{constructors} < : \text{constructor}^T}$$

Note that this is only true for the right hand side of the subtyping rule. A constructor type with a wildcard is never a subtype of a constructor type without a wildcard.

The figure Figure 4.3, “Classifier and Constructor Type Subtype Relations” shows the subtype relations defined by the preceding rules.

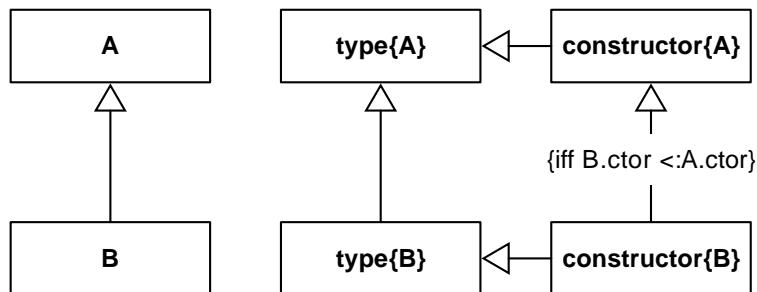


Figure 4.3. Classifier and Constructor Type Subtype Relations

Consequences:

- Overriding of static methods is possible and by using the constructor or classifier type, polymorphism for static methods is possible as well.

#### Example 4.16. Static Polymorphism

```

class A {
    static foo(): string { return "A"; }
    static bar(): string { return this.foo(); }
}
class B extends A {
    @Override
    static foo(): string { return "B"; }
}

A.bar(); // will return "A"
B.bar(); // will return "B", as foo() is called polymorphical

```

- It is even possible to refer to the constructor of an abstract class. The abstract class itself cannot provide this constructor (it only provides a type..), that is to say only concrete subclasses can provide constructors compatible to the constructor.

#### Example 4.17. Constructor of Abstract Class

```

abstract class A {}
class B extends A {}

function f(ctor: constructor{A}): A { return new ctor(); }

f(A); // not working: type{A} is not a subtype of constructor{A}.
f(B); // ok

```

Allowing wildcards on constructor type references has pragmatic reasons. The usage of constructor references usually indicates very dynamic scenarios. In some of these scenarios, e.g., in case of dynamic creation of objects in the context of generic testing or injectors, arbitrary constructors may be used. Of course, it won't be possible to check the correct new expression call in these cases – and using new expressions is prevented by N4JS if the constructor reference contains a wildcard. But other constraints, implemented by the client logic, may guarantee correct instantiation via more dynamic constructors, for example via the ECMAScript 2015 reflection API. In order to simplify these scenarios and preventing the use of `any`, wildcards are supported in constructors. Since a constructor with a wildcard cannot be used in a new expression anyway, using a classifier type is usually better than using a constructor type with wildcard.

Using wildcards on classifier types would have the same meaning as using the upper bound directly. That is, a type reference `type{? extends C}` can simply be replaced with `type{C}`, and `type{?}` with `type{any}`.

To conclude this chapter, let us compare the different types introduced above depending on whether they are used with wildcards or not:

1. having a value of type `constructor{c}` , we know we have

- a constructor function of `{c}` or a subclass of `{c}` ,
- that can be used for instantiation (i.e. the represented class is not abstract),
- that has a signature compatible to the owned or inherited constructor of `{c}` .

This means we have the constructor function of class `{c}` (but only if is non-abstract) or the constructor function of any non-abstract subclass of `{c}` with an override compatible signature to that of `{c}` 's constructor function.

2. having a value of type `constructor{? extends c}` , we know we have

- a constructor function of `{c}` or a subclass of `{c}` ,
- that can be used for instantiation (i.e. the represented class is not abstract).

So, same situation as before except that we know nothing about the constructor function's signature. However, if `{c}` has a covariant constructor, cf. [Section 5.2.5.4, “Covariant Constructors”](#), we can still conclude that we have an override compatible constructor function to that of `{c}` , because classes with covariant constructors enforce all their subclasses to have override compatible constructors.

3. have a value of type `type{? extends c}` or `type{c}` (the two types are equivalent), we know we have:

- an object representing a type (often constructor functions are used for this, e.g. in the case of classes, but could also be a plain object, e.g. in the case of interfaces),
- that represents type `{c}` or a subtype thereof,
- that cannot be used for instantiation (e.g. could be the constructor function of an abstract class, the object representing an interface, etc.).

Slightly simplified, we can say that in the first above case we can always use the value for creating an instance with `new` , in the second case only if the referenced type has a covariant constructor, cf. [Section 5.2.5.4, “Covariant Constructors”](#), and never in the third case.

#### 4.11.3. Constructors and Prototypes in ECMAScript 2015

[Figure 4.4, “Constructors and prototypes for two classes A and B in ECMAScript 2015 \(not N4JS!\)”](#) for two classes A and B in ECMAScript 2015 shows the constructors, prototypes, and the relations between them for the ECMAScript 2015 code shown in [Example 4.18, “Constructors and Prototypes”](#).

**Example 4.18. Constructors and Prototypes**

```
class A {}
class B extends A {}

var b = new B();
```



[Figure 4.4, “Constructors and prototypes for two classes A and B in ECMAScript 2015 \(not N4JS!\)”](#) shows plain ECMAScript 2015 only. Also note that `A` is defined without an `extends` clause, which is what ECMAScript 2015 calls a *base class* (as opposed to a *derived class*). The constructor of a base class always has `Function.prototype` as its prototype. If we had defined `A` as `class A extends Object {}` in the listing above, then the constructor of `A` would have `Object`'s constructor as its prototype (depicted in as a dashed red arrow), which would make a more consistent overall picture.

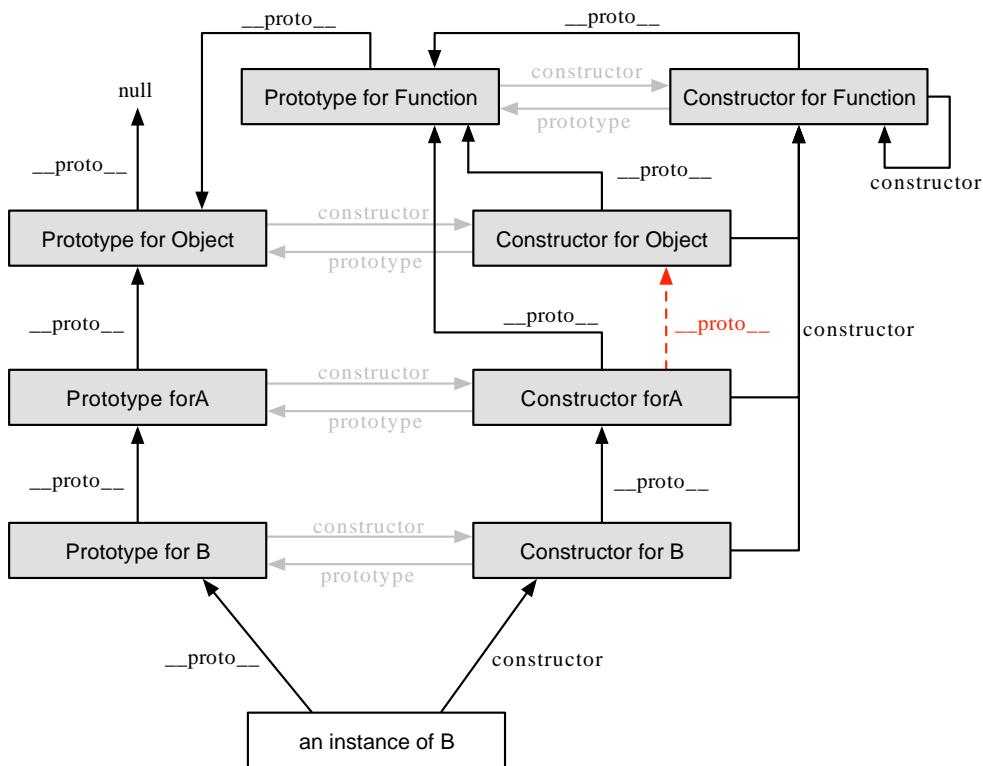


Figure 4.4. Constructors and prototypes for two classes A and B in ECMAScript 2015 (not N4JS!)

Base classes in the above sense are not available in N4JS. If an N4JS class does not provide an `extends` clause, it will implicitly inherit from built-in class `N4Object`, if it provides an `extends` clause stating `Object` as its super type, then it corresponds to what is shown in Figure 4.4, “Constructors and prototypes for two classes A and B in ECMAScript 2015 (not N4JS!)” with the red dashed arrow.

## 4.12. This Type

The `this` keyword may represent either a `this` literal (cf. Example 4.19, “This keyword and type in instance and static context”) or may refer to the `this` type. In this section, we describe the latter case.

Typical use cases of the `this` type include:

- declaring the return type of instance methods
- declaring the return type of static methods
- as formal parameter type of constructors in conjunction with use-site structural typing
- the parameter type of a function type expression, which appears as type of a method parameter
- the parameter type in a return type expression (`type {this}, constructor{this}`)
- an existential type argument inside a return type expression for methods (e.g. `ArrayList<? extends this> method(){}...`)

The precise rule where it may appear is given below in [Req-IDE-37].

The `this` type is similar to a type variable, and it is bound to the declared or inferred type of the receiver. If it is used as return type, all return statements of the methods must return the `this` keyword or a variable value implicitly inferred to a `this` type (e.g. `var x = this; return x;`).

### Simple This Type

```
class A {
    f(): this {
        return this;
    }
}
class B extends A {}

var a: A; var b: B;
a.f(); // returns something with the type of A
b.f(); // returns something with the type of B
```

`this` can be thought of as a type variable which is implicitly substituted with the declaring class (i.e. this type used in a class `{A}` actually means `<? extends A>`).

### 4.12.1. Syntax

```
ThisTypeRef returns ThisTypeRef:
    ThisTypeRefNominal | ThisTypeRefStructural;

ThisTypeRefNominal returns ThisTypeRefNominal:
    {ThisTypeRefNominal} 'this'
;

ThisTypeRefStructural returns ThisTypeRefStructural:
    typingStrategy=TypingStrategyUseSiteOperator
    'this'
    ('with' '{' ownedStructuralMembers+=TStructMember* '}')?
;
```

The keyword `this` and the type expression `this` look similar, however they can refer to different types. The type always refers to the type of instances of a class. The `this` keyword refers to the type of instances of the class in case of instance methods, but to the classifier the of the class in case of static methods. See [Section 2.1.2, "This Keyword"](#) for details.

#### Example 4.19. This keyword and type in instance and static context

Note that the following code is not working, because some usages below are not valid in N4JS. This is only to demonstrate the types.

```
class C {
    instanceMethod() {
        var c: this = this;
    }
    static staticMethod() {
        var C: type{this} = this;
    }
}
```

Structural typing and additional members in structural referenced types is described in [Section 5.3, "Structural Typing"](#).

### 4.12.2. Semantics

#### This Type

##### Requirement: IDE-37: This Type (ver. 1)

- `this` used in the context of a class is actually inferred to an existential type `? extends A` inside the class itself.
- the `this` type may only be used
  - as the type of a formal parameter of a constructor, if and only if combined with use-site structural typing.
  - at covariant positions within member declarations, except for static members of interfaces.

#### Remarks

- Due to the function subtype relation and constraints on overriding methods (in which the overriding method has to be a subtype of the overridden method), it is not possible to use the `this` type in formal parameters but only as return type. The following listing demonstrates that problem:

```
class A {
    bar(x: this): void { ... } // error
    // virtually defines: bar(x: A): void
}
class B extends A {
    // virtually defines: bar(x: B): void
}
```

As the `this` type is replaced similar to a type variable, the virtually defined method `bar` in is not override compatible with `bar` in `A`.

In case of constructors, this problem does not occur because a subclass constructor does not need to be override compatible with the constructor of the super class. Using `this` as the type of a constructor's parameter, however, would mean that you can only

create an instance of the class if you already have an instance (considering that due to the lack of method overloading a class can have only a single constructor), making creation of the first instance impossible. Therefore, `this` is also disallowed as the type of a constructor's parameter.

- The difference between the type `this` and the keyword `this` is when and how the actual type is set: the actual type of the `this` type is computed at compile (or validation) time and is always the containing type (of the member in which the type expression is used) or a subtype of that type – this is not a heuristic, this is so by definition. In contrast, the actual type of the keyword `this` is only available at runtime, while the type used at compilation time is only a heuristically-computed type, in other words, a good guess.
- The value of the `this` type is, in fact, not influenced by any `@This` annotations. Instead of using `this` in these cases, the type expressions in the `@This` annotations can be used.
- The `this` type is always bound to the instance-type regardless of the context it occurs in (non-static or static). To refer to the `this`-classifier (static type) the construct `type{this}` is used.

**Example 4.20. This type in function-type-expression**

```
class A {
    alive: boolean = true;
    methodA(func: function(this)): string {
        func(this); // applying the passed-in function
        return "done";
    }
}
```

The use of *this* type is limited to situations where it cannot be referred in mixed co- and contra-variant ways. In the following example the problem is sketched up.<sup>17</sup>

**Example 4.21. Problems with this type and type arguments**

```
// Non-working example, see problem in line 15.
class M<V> { public value: V; }

class A {
    public store: M<function(this)>; // usually not allowed, but let's assume it would be possible-----
}

class B extends A { public x=0; } // type of store is M<function(B)>

var funcA = function(a: A) {/*...something with a...*/}
var funcB = function(b: B) { console.log(b.x); }
var a: A = new A(); var b: B = new B();
b.store.value = funcA // OK, since {function(A)} <: {function(B)}
b.store.value = funcB // OK.

var a2: A = b; // OK, since B is a subtype of A
a2.store.value( a ) // RUNTIME ERROR, the types are all correct, but remember b.store.value was assigned to funcB, which
can only handle subtypes of B!
```

## 4.13. Enums

Enums are an ordered set of literals. Although enums are not true classes, they come with built-in methods for accessing value, name and type name of the enum.

In N4JS, two flavours of enumerations are distinguished: ordinary enums (N4JS) and string based enums. Ordinary enums (or in short, enums) are used while programming in N4JS. String based enums are introduced to access enumerations derived from standards, mainly developed by the W3C, in order to access the closed set of string literals defined in webIDL syntax.

### 4.13.1. Enums (N4JS)

Definition and usage of an enumeration:

```
// assume this file to be contained in a package "myPackage"
enum Color {
    RED, GREEN, BLUE
}

enum Country {
    DE : "276",
    US : "840",
    TR : "792"
}

var red: Color = Color.RED;
var us: Country = Country.US;

console.log(red.name); // --> RED
console.log(red.value); // --> RED
console.log(red.n4class.fqn); // --> myPackage.Color
console.log(red.toString()); // --> RED

console.log(us.name); // --> US
console.log(us.value); // --> 840
console.log(us.n4class.fqn); // --> myPackage.Country
console.log(us.toString()); // --> 840
```

#### 4.13.1.1. Syntax

N4EnumDeclaration <Yield>:

<sup>17</sup> The phenomenon is described in IDEBUG-263

```
=> ( {N4EnumDeclaration}
  (declaredModifiers+=N4Modifier)*
  'enum' name=BindingIdentifier<Yield>? )
  '{'
  (literals+=N4EnumLiteral (',' literals+=N4EnumLiteral)*)?
  '}';
N4EnumLiteral: name=IdentifierName (':' value=STRING)?;
```

#### 4.13.1.2. Semantics

The enum declaration  $E$  is of type `type{E}` and every enumeration is implicitly derived from `N4Enum`. There are similarities to other languages such as Java, for example, where the literals of an enum are treated as final static fields with the type of the enumeration and the concrete enumeration provides specific static methods including the literals. This leads to the following typing rules:

##### Enum Type Rules

###### Requirement: IDE-38: Enum Type Rules (ver. 1)

For a given enumeration declaration  $E$  with literals  $L$ , the following type rules are defined:

1. Every enumeration  $E$  is a subtype of the base type `N4Enum`:

$$\overline{\Gamma \vdash E < : N4Enum}$$

which itself is a subtype of `Object`:

$$\overline{N4Enum < : Object}$$

2. Every literal  $L$  of an enumeration  $E$  is of the type of the enumeration:

$$\frac{L \in E.literals}{\Gamma \vdash L:E}$$

This means that every literal is a subtype of `N4Enum` and `Object`:

$$\frac{L \in E.literals}{L < : N4Enum \wedge L < : Object}$$

The base enumeration type `N4Enum` is defined as follows:

```
/**
 * Base class for all enumeration, literals are assumed to be static constant fields of concrete subclasses.
 */
public object N4Enum {

  /**
   * Returns the name of a concrete literal
   */
  public get name(): string

  /**
   * Returns the value of a concrete literal. If no value is
   * explicitly set, it is similar to the name.
   */
  public get value(): string

  /**
   * Returns a string representation of a concrete literal, it returns
   * the same result as value()
   */
  public toString(): string

  /**
   * Returns the meta class object of this enum literal for reflection.
   * The very same meta class object can be retrieved from the enumeration type directly.
   */
  public static get n4type(): N4EnumType

  //IDE-785 this as return type in static

  /**
   * Returns array of concrete enum literals
   */
  public static get literals(): Array<? extends this>
```

```

    /**
     * Returns concrete enum literal that matches provided name,
     * if no match found returns undefined.
     */
    public static findLiteralByName(name: string): this

    /**
     * Returns concrete enum literal that matches provided value,
     * if no match found returns undefined.
     */
    public static findLiteralByValue (value: string): this
}

```

### Unique literal names

**Requirement: IDE-39: Unique literal names (ver. 1)**

\*  $\forall i, j: \text{literals}[i].name = \text{literals}[j].name \Leftrightarrow i = j$

Literal names have to be unique.

### Enum Literals are Singletons

**Requirement: IDE-40: Enum Literals are Singletons (ver. 1)**

Enum literals are singletons:

$\forall e_1, e_2, \mu(e_1) = \mu(e_2) = \text{N4EnumLiteral} \wedge \Gamma \vdash e_1 = \Gamma \vdash e_2 : e_1 == e_2 \Leftrightarrow e_1 == e_2$

#### Example 4.22. Enumeration List

Due to the common base type `N4Enum` it is possible to define generics accepting only enumeration, as shown in this example:

```

enum Color { R, G, B}

class EList<T extends N4Enum> {
    add(t: T) {}
    get(): T { return null; }
}

var colors: EList<Color>;
colors.add(Color.R);
var c: Color = colors.get();

```

### 4.13.2. String-Based Enums

In current web standards [W3C:Steen:14:Xl], definitions of enumerations are often given in webIDL syntax. While the webIDL-definition assembles a set of unique string literals as a named enum-entity, the language binding to ECMAScript refers to the usage of the members of these enumerations only. Hence, if an element of an enumeration is stored in a variable or field, passed as a parameter into a method or function or given back as a result, the actual type in JavaScript will be `string`. To provide the N4JS user with some validations regarding the validity of a statement at compile time, a special kind of subtypes of `string` are introduced: the string-based enum using the `@stringBased` annotation. (See also other string-based types like `typename<T> pathSelector<T>` and `i18nKey` in Section 4.6.4, “Primitive Pathselector and I18nKey”.)

String-based enums do not have any kind of runtime representation; instead, the transpiler will replace each reference to a literal of a string-based enum by a corresponding string literal in the output code. Furthermore, no meta-information is available for string-based enums, i.e. the `n4type` property is not available. The only exception is the static getter `literals`: it is available also for string-based enums and has the same meaning. In case of string-based enums, however, there won't be a getter used at runtime; instead, the transpiler replaces every read access to this getter by an array literal containing a string literal for each of the enum's literals.

#### String-Based Enum Type Rules

**Requirement: IDE-41: String-Based Enum Type Rules (ver. 1)**

For a string-based enum declaration  $E_S$  with literals  $L_S$  the following type rules are defined:

1. Every string-based enumeration  $E_S$  is a subtype of the base type `N4StringBasedEnum`:

$$\Gamma \vdash \text{type}\{E_S\} < : \text{N4StringBasedEnum}$$

which itself is not related to the standard enumeration type `N4Enum`

$$\begin{array}{l} \text{N4StringBasedEnum} \not< : \text{N4Enum} \\ \text{N4Enum} \not< : \text{N4StringBasedEnum} \end{array}$$

2. `N4StringBasedEnum` is a subtype of `string`

$$\text{N4StringBasedEnum} <: \text{string}$$

3. Each literal in  $L_S$  of a string-based enumeration  $E_S$  is of the type of the string-based enumeration.

$$\frac{l \in E_S L_S}{\Gamma \vdash l < : E_S}$$

4. [Req-IDE-39] also applies for `N4StringBasedEnum`.

5. [Req-IDE-40] also applies for `N4StringBasedEnum`.

6. References to string-based enums may only be used in the following places:

- a. in type annotations
- b. in property access expressions to refer to one of the enum's literals
- c. in property access expressions to read from the static getter `literals`

In particular, it is invalid to use the type of a string-based enum as a value, as in

```
@StringBased enum Color { RED, GREEN, BLUE }
var c = Color;
```

#### Example 4.23. WebIDL example

##### Gecko-Engine webIDL XMLHttpRequestResponseType as taken from [W3C:Steen:14:XL]

```
enum XMLHttpRequestResponseType {
  "",
  "arraybuffer",
  "blob",
  "document",
  "json",
  "text" //, ... and some mozilla-specific additions
}
```

Compatible Definition of this Enumeration in N4JS, provided through a runtime-library definition:

##### File in source-folder: w3c/dom/XMLHttpRequestResponseType.n4js

```
@StringBased enum XMLHttpRequestResponseType {
  vacant : "",
  arrayBuffer : "arraybuffer",
  blob : "blob",
  document : "document",
  json : "json",
  text : "text"
}
```

Usage of the enumeration in the definition files of the runtime-library. Note the explicit import of the enumeration.

##### XMLHttpRequestResponse.n4jsd

```
@@ProvidedByRuntime
import XMLHttpRequestResponseType from "w3c/dom/XMLHttpRequestResponseType";
@Global
export external public class XMLHttpRequestResponse extends XMLHttpRequestEventTarget {
  // ...
  // Setter Throws TypeError Exception
  public responseType: XMLHttpRequestResponseType;
  // ...
}
```

Client code importing the runtime-library as defined above can now use the Enumeration in a type-safe way:

##### String-Based Enumeration Usage

```
import XMLHttpRequestResponseType from "w3c/dom/XMLHttpRequestResponseType";

public function process(req: XMLHttpRequest) : void {
    if( req.responseType == XMLHttpRequestResponseType.text ) {
        // do stuff ...
    } else {
        // signal unrecognized type.
        var errMessage: req.responseType + " is not supported"; // concatenation of two strings.
        show( errMessage );
    }
}
```



# Chapter 5. Classifiers

## 5.1. N4JS Specific Classifiers

N4JS provides three new metatypes: class, interface, and enums. In this section we describe classes and interfaces. These metatypes, called *classifiers*, share some common properties which are described before type specific properties are outlined in the following sections.

All of these metatypes can be marked with type access modifiers:

```
enum N4JSTypeAccessModifier: project | public;
```

### 5.1.1. Properties

**Properties defined by syntactic elements:**

<code>annotations</code>	Arbitrary annotations, see <a href="#">Chapter 10, Annotations</a> for details.
<code>accessModifier</code>	N4JS type access modifier: <code>public</code> , or <code>project</code> ; <code>public</code> can be combined with <code>@Internal</code> ; if <code>export</code> is <code>true</code> the default is else the default is <code>private</code> .
<code>name</code>	The simple name of a classifier. If the classifier is defined by an anonymous class expression, an artificial but unique name is created. The name needs to be a valid identifier, see <a href="#">Section 3.2.2, "Valid Names"</a> .
<code>typeParams</code>	Collection of type parameters of a generic classifier; empty by default.
<code>ownedMembers</code>	Collection of owned members, i.e. methods and fields defined directly in the classifier and, if present, the explicitly defined constructor. Depending on the concrete classifier, additional constraints are defined.
<code>typingStrategy</code>	The definition-site typing strategy. By default nominal typing is used. See <a href="#">Section 5.3, "Structural Typing"</a> for details.

**The following pseudo properties are defined via annotations:**

<code>export</code>	Boolean property set to true if the <code>export</code> modifier is set. If value is true, the classifier may be accessible outside the project.
<code>final</code>	Boolean property which is set to final if annotation <code>@Final</code> is set. Also see <a href="#">Section 5.1.3.3, "Final Modifier"</a>
<code>deprecated</code>	Boolean property set to true if annotation <code>@Deprecated</code> is set.



Version 0.4, not implemented in Version 0.3

**We additionally define the following pseudo properties:**

<code>acc</code>	Type access modifier as described in <a href="#">Section 3.2, "Accessibility of Types, Top-Level Variables and Function Declarations"</a> , it is the aggregated value of the <code>accessModifier</code> and the <code>export</code> property.
<code>owned{Fields Methods Getters Setters Accessors}</code>	Filters ownedMembers by metatype, short for $x \in \text{ownedMembers}, \mu(x) = \text{Field etc.}$
<code>members</code>	Reflexive transitive closure of all members of a classifier and its super classifiers, see <a href="#">Section 5.1.2, "Common Semantics of Classifiers"</a> on how this is calculated.
<code>fields methods getters setters accessors</code>	Filters members by metatype, short for $x \in \text{members}, \mu(x) = \text{Field etc.}$
<code>superClassifiers</code>	Classes and interface may extend or implement classes or interfaces. Any class or interface extended or interface implemented is called <i>super classifier</i> . We distinguish the directly subtyped classifiers and from the transitive closure of supertypes <i>superClassifiers</i> *

### 5.1.2. Common Semantics of Classifiers

Subtyping of Classifiers

**Requirement: IDE-42: Subtyping of Classifiers** (ver. 1)

For a given type `C`, and supertypes  $superClassifiers = \{S_1, \dots, S_n\}$  directly subtyped `C`, the following constraints must be true:

1. The supertypes must be accessible to the subtype:  
 $S_1, \dots, S_n$  must be accessible to `C`.
2. All type parameters of the direct supertypes have to be bound by type arguments in the subtype and the type arguments have to be substitutable types of the type parameters.  
 $\forall 0 < i \leq k : \forall P \in S_i : \exists A \in C.typeArgs:bind(A, P) \wedge A.upperBound < :P.upperBound$
3. Wildcards may not be used as type argument when binding a supertype's type parameters.
4. A classifier cannot be directly subtyped directly multiple times:  
 $\forall S_i \forall S_j (i, j \in \{1..n\} : S_i = S_j \rightarrow i = j)$

In order to simplify the following constraints, we use the pseudo property *members* to refer to all members of a classifier. This includes all members directly declared by the classifier itself, i.e. the *ownedMember*, and all members inherited from its super classifiers. The concrete mechanisms for inheriting a member are different and further constraint (cf. [Section 5.2.8, "Redefinition of Members"](#)). A classifier only inherits its members from its direct supertypes, although the supertypes may contain members inherited from their supertypes.

## 5.1.3. Classes

### 5.1.3.1. Definition of Classes

Classes are either declared with a class declaration on top level, or they can be used as anonymous classes in expressions. The latter may have a name, which may be used for error messages and reflection.

At the current stage, class expressions are effectively disabled at least until the semantics of them are finalized in ECMAScript 6.

In N4JS (as in many other languages) multi-inheritance of classes is not supported. Although the `diamond problem` (of functions being defined in both superclasses) could be solved via union and intersection types, this would lead to problems when calling these super implementations. This is particularly an issue due to JavaScript not supporting multiple prototypes.<sup>1</sup> Interfaces, however, allow for multi-inheritance. Since the former can also define functions with bodies, this is not a hard restriction.

#### 5.1.3.1.1. Syntax

##### Syntax N4 Class Declaration and Expression

```
N4ClassDeclaration <Yield>:
  =>
    {N4ClassDeclaration}
    annotations+=Annotation*
    (declaredModifiers+=N4Modifier)*
    'class' typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield>?
  )
  TypeVariables?
  ClassExtendsClause<Yield>?
  Members<Yield>
;

N4ClassExpression <Yield>:
  {N4ClassExpression}
  'class' name=BindingIdentifier<Yield>?
  ClassExtendsClause<Yield>?
  Members<Yield>;

fragment ClassExtendsClause <Yield>*:
  'extends' (
    =>superClassRef=ParameterizedTypeRefNominal ('implements' ClassImplementsList)?
    | superClassExpression=LeftHandSideExpression<Yield>
  )
  | 'implements' ClassImplementsList
```

<sup>1</sup> E.g., for given `class A{ foo(A):A{}}` `class B{ foo(B):B{}}`, a class C could be defined as `class C{ foo(union{A,B}):intersection{A,B}{}}`. In this case it would then be a syntactical problem (and even worse - a conceptual problem) of how to call the super methods defined in A and B from C.

```

;
fragment ClassImplementsList*:
    implementedInterfaceRefs+=ParameterizedTypeRefNominal
    ('.' implementedInterfaceRefs+=ParameterizedTypeRefNominal)*
;

fragment Members <Yield>*:
    '{'
    ownedMembers+=N4MemberDeclaration<Yield>*
    '}'
;

```

### 5.1.3.1.2. Properties

These are the properties of class, which can be specified by the user: Syntax N4 Class Declaration and Expression

<b>abstract</b>	Boolean flag indicating whether class may be instantiable; default is <code>false</code> , see <a href="#">Section 5.1.3.4, “Abstract Classes”</a> .
<b>external</b>	Boolean flag indicating whether class is a declaration without implementation or with an external (non-N4JS) implementation; default is <code>false</code> , see <a href="#">Section 5.3.2, “Definition Site Structural Typing”</a> .
<b>defStructural</b>	Boolean flag indicating whether subtype relation uses nominal or structural typing, see <a href="#">Section 5.3.2, “Definition Site Structural Typing”</a> for details.
<b>superType/sup</b>	The type referenced by <i>superType</i> is called direct superclass of a class, and vice versa the class is a direct subclass of <i>superType</i> . Instead of <i>superType</i> , we sometimes simply write <i>sup</i> . The derived set <i>sup<sup>+</sup></i> is defined as the transitive closures of all direct and indirect superclasses of a class. If no supertype is explicitly stated, classes are derived from <code>N4Object</code> .
<b>implementedInterfaces/interfaces</b>	of interfaces directly <i>implemented</i> by the class; empty by default. Instead of <code>implementedInterfaces</code> , we simply write <code>interfaces</code> .
<b>ownedCtor</b>	Explicit constructor of a class (if any), see <a href="#">Section 4.11, “Constructor and Classifier Type”</a> .

And we additionally define the following pseudo properties:

<b>ctor</b>	Explicit or implicit constructor of a class, see <a href="#">Section 4.11, “Constructor and Classifier Type”</a> .
<b>fields</b>	Further derived properties for retrieving all methods (property <i>methods</i> ), fields (property <i>fields</i> ), static members (property <i>staticOwnedMembers</i> ), etc. can easily be added by filtering properties <i>members</i> or <i>ownedMembers</i> .

### 5.1.3.1.3. Type Inference

The type of a class declaration or class expression *C* (i.e., a class definition in general) is of type `constructor{C}` if it is not abstract, that is if it can be instantiated. If it is abstract, the type of the definition simply is `type{C}`:

$$\frac{\neg C:\text{Abstract}}{I \vdash C:\text{constructor}\{C\}}$$

$$\frac{C:\text{Abstract}}{I \vdash C:\text{type}\{C\}}$$

Structural and Nominal Supertypes

**Requirement: IDE-43: Structural and Nominal Supertypes (ver. 1)**

The type of supertypes and implemented interfaces is always the nominal type, even if the supertype is declared structurally.

### 5.1.3.2. Semantics

This section deals with the (more or less) type-independent constraints on classes.

Class expressions are not fully supported at the moment.

Transitive closure of members

**Definition:** Transitive closure of members

The reflexive transitive closure of members of a class is indirectly defined by the override and implementation constraints defined in [Section 5.2.8, “Redefinition of Members”](#).

Note that since overloading is forbidden, the following constraint is true <sup>2</sup>:

$$\forall m_1, m_2 \in members: m_1.name = m_2.name \Leftrightarrow m_1 = m_2 \vee accessorPair(m_1, m_2)$$

Remarks: Class and method definition is quite similar to the proposed ECMAScript version 6 draft [ECMA15a(p.S13.5)], except that an N4 class and members may contain

- annotations, abstract and access modifiers
- fields
- types
- implemented interfaces

Note that even `static` is used in ECMAScript 6.

Mixing in members (i.e. interface's methods with default implementation or fields) is similar to mixing in members from roles as defined in [Dart13a(p.S9.1)]. It is also similar to default implementations in Java 8 [Gosling15a]. In Java, however, more constraints exist, (for example, methods of interfaces must be public).

#### Example 5.1. Simple Class

This first example shows a very simple class with a field, a constructor and a method.

```
class C {
    data: any;

    constructor(data: any) {
        this.data = data;
    }

    foo(): void { }
}
```

#### Example 5.2. Extend and implement

The following example demonstrate how a class can extend a superclass and implement an interface.

```
interface I {
    foo(): void
}
class C{}
class X extends C implements I {
    @Override
    foo(): void {}
}
```

A class `C` is a subtype of another classifier `S` (which can be a class or interface) if the other classifier `S` is (transitively) contained in the supertypes (superclasses or implemented interfaces) of the class:

$$\frac{\left. \begin{array}{c} left = right \\ \Gamma \vdash TClass \; left < : [TClass] \; right \end{array} \right. \text{[shortcut]} \quad \Gamma \vdash left.superType.declaredType < : right}{\Gamma \vdash TClass \; left < : [TClass] \; right}$$

#### Implicit Supertype of Classes

##### Requirement: IDE-44: Implicit Supertype of Classes (ver. 1)

1. The implicit supertype of all classes is `N4Object`. All classes with no explicit supertype are inherited from `N4Object`. 2. If the supertype is explicitly set to `Object`, then the class is not derived from `N4Object`. Meta-information is created similar to an `N4Object`-derived class. Usually, there is no reason to explicitly derive a class from `Object`. 3. External classes are implicitly derived from , unless they are annotated with `@N4JS` (cf. Section 13.2, “External Declarations”).

### 5.1.3.3. Final Modifier

Extensibility refers to whether a given classifier can be subtyped. Accessibility is a prerequisite for extensibility. If a type cannot be seen, it cannot be subclassed. The only modifier influencing the extensibility directly is the annotation `@Final`, which prevents all

<sup>2</sup> `accessorPair(m1, m2)` is defined as follows:  $(\mu(m_1) = \text{getter} \wedge \mu(m_2) = \text{setter}) \vee (\mu(m_1) = \text{setter} \wedge \mu(m_2) = \text{getter})$

subtyping. The following table shows how to prevent other projects or vendors from subtyping by also restricting the accessibility of the constructor:

**Table 5.1. Extensibility of Types**

Type <code>C</code> Settings	Subclassed in		
	Project	Vendor	World
<code>C.final</code>	no	no	no
<code>C.ctor.accessModifier=\lenum{project}</code>	yes	no	no
<code>C.ctor.accessModifier=\lenum{public@Internal}</code>	yes	yes	no

Since interfaces are always to be implemented, they must not be declared final.

#### 5.1.3.4. Abstract Classes

A class with modifier `abstract` is called an *abstract class* and has its `abstract` property set to true. Other classes are called *concrete classes*.

##### Abstract Class

###### Requirement: IDE-45: Abstract Class (ver. 1)

1. A class `C` must be declared abstract if it owns or inherits one or more abstract members and neither `C` nor any interfaces implemented by `C` implements these members. A concrete class has to, therefore, implement all abstract members of its superclasses' implemented interfaces. Note that a class may implement fields with field accessors and vice versa.
2. An abstract class may not be instantiated.
3. An abstract class cannot be set to final (with annotation `@Final`).

##### Abstract Member

###### Requirement: IDE-46: Abstract Member (ver. 1)

1. A member declared as abstract must not have a method body (in contrary a method not declared as abstract have to have a method body).
2. Only methods, getters and setters can be declared as abstract (fields cannot be abstract).
3. It is not possible to inherit from an abstract class which contains abstract members which are not visible in the subclass.
4. An abstract member cannot be set to final (with annotation `@Final`).
5. Static members may not be declared abstract.

#### 5.1.3.5. Non-Instantiable Classes

To make a class non-instantiable outside a defining compilation unit, i.e. disallow creation of instances for this class, simply declare the constructor as private. This can be used for singletons.

#### 5.1.3.6. Superclass

##### Superclass

###### Requirement: IDE-47: Superclass (ver. 1)

For a class `C` with a supertype `S = C.sup`, the following constraints must hold;

- `C.sup` must reference a class declaration `s`
- `s` must be extendable in the project of `C`
- `C ≠ C.sup+`
- All abstract members in `s` must be accessible from `C`:

$$\forall M \in S.members : M.abstract =$$

`M` is accessible from `C`.

Note that `M` need not be an owned member of `s` and that this constraint applies even if `C` is abstract).

All members of superclasses become members of a class. This is true even if the owning classes are not directly accessible to a class. The member-specific access control is not changed.

## 5.1.4. Interfaces

### 5.1.4.1. Definition of Interfaces

#### 5.1.4.1.1. Syntax

##### Syntax N4 Interface Declaration

```
N4InterfaceDeclaration <Yield>:
  => (
    {N4InterfaceDeclaration}
    annotations+=Annotation*
    (declaredModifiers+=N4Modifier)*
    'interface' typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield>?
  )
  TypeVariables?
  InterfaceImplementsList?
  Members<Yield>
;

fragment InterfaceImplementsList*:
  'implements' superInterfaceRefs+=ParameterizedTypeRefNominal
  (',' superInterfaceRefs+=ParameterizedTypeRefNominal)*
;
```

#### 5.1.4.1.2. Properties

These are the additional properties of interfaces, which can be specified by the user:

**superInterfaces** Collection of interfaces extended by this interface; empty by default. Instead of `superInterfaces`, we simply write `interfaces`.

#### 5.1.4.1.3. Type Inference

The type of an interface declaration `I` is of type `type{I}`:

$$\Gamma \vdash I : type\{I\}$$

#### 5.1.4.1.4. Semantics

Interfaces are used to describe the public API of a classifier. The main requirement is that the instance of an interface, which must be an instance of a class since interfaces cannot have instances, provides all members declared in the interface. Thus, a (concrete) class implementing an interface must provide implementations for all the fields, methods, getters and setters of the interface (otherwise it the class must be declared abstract). The implementations have to be provided either directly in the class itself, through a superclass, or by the interface if the member has a default implementation.

A field declaration in an interface denotes that all implementing classes can either provide a field of the same name and the same(!) type or corresponding field accessors. If no such members are defined in the class or a (transitive) superclass, the field is mixed in from the interface automatically. This is also true for the initializer of the field.

All instance methods, getters and setters declared in an interface are implicitly abstract if they do not provide a default implementation. The modifier `abstract` is not required, therefore, in the source code. The following constraints apply:

##### Interfaces

###### Requirement: IDE-48: Interfaces (ver. 1)

For any interface `I`, the following must hold:

1. Interfaces may not be instantiated.
2. Interfaces cannot be set to final (with annotation `@Final`):  $\neg I.final$ .
3. Members of an interface must not be declared private. The default access modifier in interfaces is the the type's visibility or `project`, if the type's visibility is `private`.
4. Members of an interface, except methods, must not be declared `@Final`:

$$\forall m \in I.member : m.final \rightarrow m \in I.methods$$



not allowing field accessors to be declared final was a deliberate decision, because it would complicate the internal handling of member redefinition; might be reconsidered at a later time

##### 5. The literal may not be used in the initializer expression of a field of an interface.

This restriction is required, because the order of implementation of these fields in an implementing class cannot be guaranteed. This applies to both instance and static fields in interfaces, but in case of static fields, `this` is also disallowed due to [Section 5.2.7.3, “Static Members of Interfaces”](#).

It is possible to declare members in interfaces with a smaller visibility as the interface itself. In that case, clients of the interface may be able to use the interface but not to implement it.

In order to simplify modeling of runtime types, such as elements, interfaces do not only support the notation of static methods but constant data fields as well. Since [IDL \[OMG14a\]](#) is used to describe these elements in specifications (and mapped to JavaScript via rules described in [\[W3C12a\]](#)) constant data fields are an often-used technique there and they can be modeled in N4JS 1:1.

As specified in [\[Req-IDE-56\]](#), interfaces cannot contain a constructor i.e.

$\forall m \in I.\text{ownedMethods}: m.name \neq \text{constructor}$ .

#### Example 5.3. Simple Interfaces

The following example shows the syntax for defining interfaces. The second interface extends the first one. Note that methods are implicitly defined abstract in interfaces.

```
interface I {
    foo(): void
}
interface I2 extends I {
    someText: string;
    bar(): void
}
```

If a classifier `C` implements an interface `I`, we say `I` is *implemented by C*. If `C` redefines members declared in `I`, we say that these members are *implemented by C*. Members not redefined by `C` but with a default implementations are *mixed in* or *consumed* by `C`. We call `C` the *implementor*.

Besides the general constraints described in [Section 5.1.2, “Common Semantics of Classifiers”](#), the following constraints must hold for extending or implementing interfaces:

#### Extending Interfaces

##### Requirement: IDE-49: Extending Interfaces (ver. 1)

For a given type `I`, and  $\{I_1, \dots, I_n\}$  directly extended by `I`, the following constraints must be true:

1. Only interfaces can extend interfaces:  $I, I_1, \dots, I_n$  must be interfaces.

2. An interface may not directly extend the same interface more than once:

$$I_i = I_j \rightarrow i = j \text{ for any } i, j \in \{1 \dots n\}.$$

3. An interface may (indirectly) extend the same interface `J` more than once only if

a. `J` is not parameterized, or

b. in all cases `J` is extended with the same type arguments for all invariant type parameters.

Note that for type parameters of `J` that are declared covariant or contravariant on definition site, different type arguments may be used.

4. All abstract members in  $I_i, i \in \{1, \dots, n\}$ , must be accessible from `I`:

$$\forall i \in \{1, \dots, n\} : M \in I_i.\text{members} \wedge M.\text{abstract} \rightarrow M \text{ is accessible from } I.$$

Note that `M` need not be an owned member of  $I_i$ .

#### Implementing Interfaces

##### Requirement: IDE-50: Implementing Interfaces (ver. 1)

For a given type `C`, and  $\{I_1, \dots, I_n\}$  directly implemented by `C`, the following constraints must be true:

1. Only classes can implement interfaces: `C` must be a Class.
2. A class can only implement interfaces:  $I_1, \dots, I_n$  must be interfaces.
3. A class may not directly implement the same interface more than once:  
 $I_i = I_j = i = j$  for any  $i, j \in \{1, \dots, n\}$ .
4. A class may (indirectly) implement the same interface  $J$  more than once only if

- a.  $J$  is not parameterized, or
- b. in all cases  $J$  is implemented with the same type arguments for all invariant type parameters.

Note that for type parameters of  $J$  that are declared covariant or contravariant on definition site, different type arguments may be used.

5. All abstract members in  $I_i, i \in \{1, \dots, n\}$ , must be accessible from `C`:

$\forall i \in \{1, \dots, n\} : M \in I_i.\text{members} \wedge M.\text{abstract} \rightarrow M$  is accessible from `C`.

Note that `M` need not be an owned member of  $I_i$ .

For default methods in interfaces, see [Section 5.2.4, “Default Methods in Interfaces”](#).

### 5.1.5. Generic Classifiers

Classifiers can be declared generic by defining a type parameter via `type-param`.

#### Generic Classifiers

##### Definition: Generic Classifiers

A generic classifier is a classifier with at least one type parameter. That is, a given classifier `C` is generic if and only if  $|C.typeParams| \geq 1$ .

If a classifier does not define any type parameters, it is not generic, even if its superclass or any implemented interface is generic.

The format of the type parameter expression is described in [Section 4.4.3, “Parameterized Types”](#). The type variable defined by the type parameter’s type expression can be used just like a normal type inside the class definition.

If using a generic classifier as type of a variable, it may be parameterized. This is usually done via a type expression (cf. [Section 4.4.3, “Parameterized Types”](#)) or via `typearg` in case of supertypes. If a generic classifier defines multiple type variables, these variables are bound in the order of their definition. In any case, all type variables have to be bound. That means in particular that raw types are not allowed. (cf [Section 4.4.3, “Parameterized Types”](#) for details).

If a generic classifier is used as super classifier, the type arguments can be type variables. Note that the type variable of the super classifier is not lifted, that is to say that all type variables are to be explicitly bound in the type references used in the `extend`, `with`, or `implements` section using `typearg`. If a type variable is used in `typearg` to bound a type variable of a type parameter, it has to fulfil possible type constraints (upper/lower bound) specified in the type parameter.

#### Example 5.4. Generic Type Definition and Usage as Type of Variable

This example demonstrates how to define a generic type and how to refer to it in a variable definition.

```
export class Container<T> {
    private item: T;

    getItem(): T {
        return this.item;
    }

    setItem(item: T): void {
        this.item = item;
    }
}
```

This type can now be used as a type of a variable as follows

```
import Container from "p/Container"
```

```
var stringContainer: Container<string> = new Container<string>();
stringContainer.setItem("Hello");
var s: string = stringContainer.getItem();
```

In line 3, the type variable `T` of the generic class `Container` is bound to `string`.

**Example 5.5. Binding of type variables with multiple types**

For a given generic class `G`

```
class A{}
class B{}
class C extends A{}

class G<S, T extends A, U extends B> {
```

the variable definition

```
var x: G<Number, C, B>;
```

would bind the type variables as follows:

<code>S</code>	<code>Number</code>	Bound by first type argument, no bound constraints defined for <code>s</code> .
<code>T</code>	<code>C</code>	Bound by second type argument, <code>C</code> must be a subtype of <code>B</code> in order to fulfill the type constraint.
<code>U</code>	<code>B</code>	Bound by third type argument, <code>extends</code> is reflexive, that is <code>B</code> fulfills the type constraint.

### Generic Superclass, Type Argument with Type Variable

**Requirement: IDE-51: Generic Superclass, Type Argument with Type Variable (ver. 1)**

For a given generic superclass `SuperClass`

```
class SuperClass<S, T extends A, U extends B> {};
```

and a generic subclass `SubClass`

```
class SubClass<X extends A> extends SuperClass<Number, X, B> {...};
```

the variable definition

```
var s: SubClass<C>;
```

would bind the type variables as follows:

TypeVariable	Bound to	Explanation
<code>SuperClass.S</code>	<code>Number</code>	Type variable <code>s</code> of supertype <code>SuperClass</code> is bound to <code>Number</code> .
<code>SuperClass.T</code>	<code>SubClass.X=C</code>	Type variable <code>T</code> of supertype <code>SuperClass</code> is bound to type variable <code>X</code> of <code>SubClass</code> . It gets then indirectly bound to <code>C</code> as specified by the type argument of the variable definition.
<code>SuperClass.U</code>	<code>B</code>	Type variable <code>U</code> of supertype <code>SuperClass</code> is auto-bound to <code>C</code> as no explicit binding for the third type variable is specified.
<code>SubClass.X</code>	<code>C</code>	Bound by first type argument specified in variable definition.

## 5.1.6. Definition-Site Variance

In addition to use-site declaration of variance in the form of Java-like wildcards, N4JS provides support for definition-site declaration of variance as known from languages such as C# and Scala.

The *variance* of a parameterized type states how its subtyping relates to its type arguments' subtyping. For example, given a parameterized type `G<T>` and plain types `A` and `B`, we know

- if `G` is **covariant** w.r.t. its parameter `T`, then

$B < : A \rightarrow G < B > < : G < A >$

- if  $G$  is **contravariant** w.r.t. its parameter  $T$ , then

$B < : A \rightarrow G < A > < : G < B >$

- if  $G$  is **invariant** w.r.t. its parameter  $T$ , then

$B < : A \rightarrow G < A > < : G < B >$

$B < : A \rightarrow G < A > < : G < B >$

Note that variance is declared per type parameter, so a single parameterized type with more than one type parameter may be, for example, covariant w.r.t. one type parameter and contravariant w.r.t. another.

Strictly speaking, a type parameter/variable itself is not co- or contravariant;

however, for the sake of simplicity we say  $T$  is **covariant** as a short form for  $G$  is covariant with respect to its type parameter  $T$  (for contravariant and invariant accordingly).

To declare the variance of a parameterized classifier on definition site, simply add keyword `in` or `out` before the corresponding type parameter:

```
class ReadOnlyList<out T> { // covariance
    // ...
}

interface Consumer<in T> { // contravariance
    // ...
}
```

In such cases, the following constraints apply.

## Definition-Site Variance

### Requirement: IDE-174: Definition-Site Variance (ver. 1)

Given a parameterized type with a type parameter  $T$ , the following must hold:

1.  $T$  may only appear in variance-compatible positions:

- a. if  $T$  is declared on definition site to be **covariant**, then it may only appear in covariant positions within the type's non-private member declarations.
- b. if  $T$  is declared on definition site to be **contravariant**, then it may only appear in contravariant positions within the type's non-private member declarations.
- c. if  $T$  is **invariant**, i.e. neither declared covariant nor declared contravariant on definition site, then it may appear in any position (where type variables are allowed).

Thus, no restrictions apply within the declaration of private members and within the body of field accessors and methods.

2. definition-site variance may not be combined with incompatible use-site variance:

- a. if  $T$  is declared on definition site to be **covariant**, then no wildcard with a **lower** bound may be provided as type argument for  $T$ .
- b. if  $T$  is declared on definition site to be **contravariant**, then no wildcard with an **upper** bound may be provided as type argument for  $T$ .
- c. if  $T$  is **invariant**, i.e. neither declared covariant nor declared contravariant on definition site, then any kind of wildcard may be provided as type argument.

Unbounded wildcards are allowed in all cases.

### Example 5.6. Use-site declaration of variance

For illustration purposes, let's compare use-site and definition-site declaration of variance. Since use-site variance is more familiar to the Java developer, we start with this flavor.

```
class Person {
    name: string;
}
```

```

class Employee extends Person {}

interface List<T> {
    add(elem: T)
    read(idx: int): T
}

function getNameOfFirstPerson(list: List<? extends Person>): string {
    return list.read(0).name;
}

```

Function `getNameOfFirstPerson` below takes a list and returns the name of the first person in the list. Since it never adds new elements to the given list, it could accept `List`s of any subtype of `Person`, for example a `List<Employee>`. To allow this, its formal parameter has a type of `List<? extends Person>` instead of `List<Person>`. Such use-site variance is useful whenever an invariant type, like `List` above, is being used in a way such that it can be treated as if it were co- or contravariant.

Sometimes, however, we are dealing with types that are inherently covariant or contravariant, for example an `ImmutableList` from which we can only read elements would be covariant. In such a case, use-site declaration of variance is tedious and error-prone: we would have to declare the variance wherever the type is being used and would have to make sure not to forget the declaration or otherwise limit the flexibility and reusability of the code (for example, in the above code we could not call `getNameOfFirstPerson` with a `List<Employee>`).

The solution is to declare the variance on declaration site, as in the following code sample:

```

interface ImmutableList<out T> {
    // add(elem: T) // error: such a method would now be disallowed
    read(idx: int): T
}

function getNameOfFirstPerson2(list: ImmutableList<Person>): string {
    return list.read(0).name;
}

```

Now we can invoke `getNameOfFirstPerson2` with a `List<Employee>` even though the implementor of `getNameOfFirstPerson2` did not add a use-site declaration of covariance, because the type `ImmutableList` is declared to be covariant with respect to its parameter `T`, and this applies globally throughout the program.

## 5.2. Members

A member is either a method (which may be a special constructor function), a data field, or a getter or a setter. The latter two implicitly define an accessor field. Similar to object literals, there must be no data field with the same name as a getter or setter.

(overriding, implementation and consumption) is described in Section 5.2.8, “Redefinition of Members”.

### 5.2.1. Syntax

#### Syntax N4JS member access modifier

```

enum N4JSMemberAccessModifier: private | project | protected | public;

N4MemberDeclaration: N4MethodDeclaration | N4FieldDeclaration | N4GetterDeclaration | N4SetterDeclaration;

```

#### 5.2.1.1. Properties

Members share the following properties:

<code>annotations</code>	Arbitrary annotations, see Chapter 10, <i>Annotations</i> for details.
<code>accessModifier</code>	N4JS member access modifier: <code>private</code> , <code>project</code> , <code>protected</code> , or <code>public</code> ; the latter two can be combined with <code>@Internal</code> ; default is <code>project</code> for classes and private interfaces. For a non-private interface defaults to the interface's visibility.
<code>name</code>	The simple name of the member, that is an identifier name (cf. Section 3.2.2, “Valid Names”).
<code>static</code>	Boolean property to distinguish instance from classifier members, see Section 5.2.7, “Static Members”.

The following pseudo properties are defined via annotations:

`deprecated` Boolean property set to true if annotation `@Deprecated` is set.<sup>3</sup>

<sup>3</sup> not yet implemented

And we additionally define the following pseudo properties:

<code>acc</code>	Member access modifier as described in <a href="#">Section 3.2.1, “Accessibility of Members”</a> , it is the aggregated value of the <i>accessModifier</i> and the <i>export</i> property.
<code>owner</code>	Owner classifier of the member.
<code>typeRef</code>	Type of the member—this is the type of a field or the type of the method which is a function type (and not the return type).
<code>assignability</code>	Enumeration, may be one of the following values:
<code>set</code>	Member may only be set, i.e. it could only be used on the left hand side of an assignment.
<code>get</code>	Member may only be retrieved, i.e. it could only be used on the right hand side of an assignment. This is the default setting for methods.
<code>any</code>	Member may be set or retrieved, i.e. it could only be used on the left or right hand side of an assignment. This is the default setting for fields.



*assignability* is related but not equal to writable modifiers used for fields. We define a partial order on this enumeration as follows:

$< (l, r) ::= \{(set, any), (get, any)\}$

`abstract` All members have a flag *abstract*, which is user-defined for methods, getters and setter, but which is always false for fields.

The following pseudo property is set to make fields compatible with properties of an object literal, however it cannot be changed:

`configurable` Boolean flag reflecting the property descriptor *configurable*, this is always set to false for members.

## 5.2.2. Semantics

The members of a given classifier `C` must be named such that the following constraints are met:

### Member Names

#### Requirement: IDE-52: Member Names (ver. 1)

. The name of a member is given as an identifier, a string literal, a numeric literal, or as a computed property name with a compile-time expression (see [Section 8.5, “Compile-Time Expressions”](#)). In particular, string literals, e.g. `['myProp']`, built-in symbols, e.g. `[Symbol.iterator]`, and literals of `@StringBased` enums are all valid computed property names. . No two members may have the same name, except one is static and the other is non-static:

+

$\forall m_1, m_2 \in C.ownedMembers, m_1 \neq m_2 : m_1.name \neq m_2.name \vee m_1.static \neq m_2.static$

1. The member name must be a valid identifier name, see [Identifier Grammar](#).

Thus, *overloading* of methods is not supported <sup>4</sup> and no field may have the same name as a method. However, *overriding* of methods, getters, and setters are possible, see [Section 5.2.8, “Redefinition of Members”](#). Static members may also have the same name as non-static members.<sup>5</sup>

The dollar character `$` is not allowed for user-defined member identifiers as the dollar sign is used for rewriting private members.

## 5.2.3. Methods

Methods are simply JavaScript functions. They are defined similarly to methods as proposed in [\[ECMA15a\(p.S13.5\)\]](#) except for the type information and some modifiers.

### 5.2.3.1. Syntax

#### Syntax Method Declaration

<sup>4</sup> In order to emulate method overloading, union types are to be used.

<sup>5</sup> [\[ECMA15a\(p.p214\)\]](#), `ClassBody : ClassElementList` indicates that it is possible to have the same name for instance and static members.

```

N4MethodDeclaration <Yield>:
  => ({N4MethodDeclaration}
    annotations+=Annotation*
    accessModifier=N4JSMemberAccessModifier?
    (abstract?='abstract' | static?='static')?
    TypeVariables?
    (
      generator?='*' LiteralOrComputedPropertyName<Yield> -> MethodParamsReturnAndBody <Generator=true>
      | AsyncNoTrailingLineBreak LiteralOrComputedPropertyName<Yield> -> MethodParamsReturnAndBody <Generator=false>
    )
  ) ';'?
;

fragment MethodParamsAndBody <Generator>*:
  StrictFormalParameters<Yield=Generator>
  (body=Block<Yield=Generator>)?
;

fragment MethodParamsReturnAndBody <Generator>*:
  StrictFormalParameters<Yield=Generator>
  ('::' returnTypeRef=TypeRef)?
  (body=Block<Yield=Generator>)?
;

fragment LiteralOrComputedPropertyName <Yield>*:
  name=IdentifierName | name=STRING | name=NumericLiteralAsString
  | '[' (=>((name=SymbolLiteralComputedName<Yield> | name=StringLiteralAsName) ')') |
  computeNameFrom=AssignmentExpression<In=true, Yield> ')'
;

symbolLiteralComputedName <Yield>:
  BindingIdentifier<Yield> ('.' IdentifierName)?
;

BindingIdentifier <Yield>:
  IDENTIFIER
  | <!Yield> 'yield'
  | N4Keyword
;

IdentifierName: IDENTIFIER | ReservedWord | N4Keyword;
NumericLiteralAsString: DOUBLE | INT | OCTAL_INT | HEX_INT | SCIENTIFIC_INT;
StringLiteralAsName: STRING;

fragment AsyncNoTrailingLineBreak *: (declaredAsync?='async' NoLineTerminator)?; // See Asynchronous Functions

fragment StrictFormalParameters <Yield>*:
  '(' (fpars+=FormalParameter<Yield> ',' fpars+=FormalParameter<Yield>)*?) ')'
;

FormalParameter <Yield>:
  {FormalParameter} BindingElementFragment<Yield>
;

fragment BindingElementFragment <Yield>*:
  (=> bindingPattern=BindingPattern<Yield>
  | annotations+=Annotation*
  (
    variadic?='...'? name=BindingIdentifier<Yield> ColonSepTypeRef?
  )
  )
  ('=' initializer=AssignmentExpression<In=true, Yield>)?
;

fragment ColonSepTypeRef*:
  ':' declaredTypeRef=TypeRef
;

```

### 5.2.3.2. Properties

Methods have all the properties of members and the following additional properties can be explicitly defined:

<b>abstract</b>	Method is declared but not defined.
<b>typePars</b>	Collection of type parameters of a generic method; empty by default.
<b>returnTypeRef</b>	Return type of the method, default return type is <i>Void</i> . The type of the method as a member of the owning classifier is not the method's return type but is instead a function type.
<b>fpars</b>	List of formal parameters, may be left empty.

**body** The body of the method (this is not available in the pure types model)

The following pseudo properties are defined via annotations:

**final** Boolean flag set to true if annotation `@Final` is set. The flag indicates that method must not be overridden in subclasses; see [Section 5.2.3.4, “Final Methods”](#).

**declaresOverride** Flag set to true if annotation `@Overrides` is set. The flag indicates that method must override a method of a superclass; see [Section 5.2.8.1, “Overriding of Members”](#).

Additionally, we define the following pseudo properties:

**overrides** True if method overrides a super method or implements an interface method, false otherwise.

**typeRef** Type of the method. This is, in fact, a function type (and not the return type).

The following pseudo property is set to make methods compatible with properties of an object literal, however it cannot be changed:

**enumerable** Boolean flag reflecting the property descriptor *enumerable*, this is always set to false for methods.

### 5.2.3.3. Semantics

Since methods are ECMAScript functions, all constraints specified in [Section 6.1, “Function Type”](#) apply to methods as well. This section describes default values and function type conformance which is required for overriding and implementing methods.

In addition, method declarations and definitions have to comply with the constraints for naming members of classifiers (cf. [\[Req-IDE-52\]](#)) and with the constraints detailed in the following sections on final methods ([Section 5.2.3.4, “Final Methods”](#)), abstract methods ([Section 5.2.3.5, “Abstract Methods”](#)) and method overriding and implementation ([Section 5.2.8.1, “Overriding of Members”](#), [Section 5.2.8.2, “Implementation of Members”](#)).

The following constraints are defined for methods in ECMAScript 6 [[ECMA15a\(p.207\)](#)]

#### Method Definition ECMAScript 6

##### Requirement: IDE-53: Method Definition ECMAScript 6 (ver. 1)

\* It is a Syntax Error if any element of the BoundNames of StrictFormalParameters also occurs in the VarDeclaredNames of FunctionBody. \* It is a Syntax Error if any element of the BoundNames of StrictFormalParameters also occurs in the LexicallyDeclaredNames of FunctionBody.

Methods – like functions – define a variable execution environment and therefore provide access to the actual passed-in parameters through the implicit `arguments` variable inside of their bodies (c.f. [Section 6.1.4, “Arguments Object”](#)).

Methods are similar to function definitions but they must not be assigned to or from variables. The following code issues an error although the type of the method would be compatible to the type of the variable `v`:

```
class C {
    m(): void {}
}
var v: {function():void} = new C().m;
```

#### Method Assignment

##### Requirement: IDE-54: Method Assignment (ver. 1)

. In contrast to ECMAScript 2015, methods are defined as readonly, that is, it is not possible to dynamically re-assign a property defined as method with a new value. This is because assigning or re-assigning a method breaks encapsulation. Methods are the [Appendix A, Acronyms](#) of a class, their implementation is internal to the class. . When assigning a method to a variable, a warning is issued since this would lead to an detached this reference inside the method when it is called without explicitly providing the receiver. No warning is issued only if it is guaranteed that no problems will occur: .. The method's body can be determined at compile time (i.e., it has been declared `@Final`) and it lacks usages of `this` or `super`. This is true for instance and static methods. .. The method is the constructor.



The following code demonstrates problems arising when methods are assigned to variables in terms of function expressions. Given are two classes and instances of each class as follows:

```

class C {
    m(): void { }
    static k(): void {}
}
class D extends C {
    @Override m(): void { this.f() }
    f(): void {}

    @Override static k(): void { this.f() }
    static f(): void {}
}
var c: C = new C();
var d: C = new D(); // d looks like a C

```

Assigning an instance method to a variable could cause problems, as the method assumes this to be bound to the class in which it is defined. This may work in some cases, but will cause problems in particular in combination with method overriding:

```

var v1: {@This(C)function():void} = c.m;
var v2: {@This(C)function():void} = d.m;

v1.call(c);
v2.call(c);

```

Calling `c.m` indirectly via `v1` with `c` as this object will work. However, it won't work for `v2`: the method is overridden in `D`, and the method in expects other methods available in `D` but not in `C`. That is, the last call would lead to a runtime error as method `f` which is called in `D.m` won't be available.

The same scenario occurs in case of static methods if they are retrieved polymorphically via the variables of type `constructor{C}`:

```

var ctor: constructor{C} = C;
var dtor: constructor{C} = D;

var v3: {@This(constructor{C})function():void} = ctor.k;
var v4: {@This(constructor{C})function():void} = dtor.k;

```

In both cases, the problem could be solved by restricting these kinds of assignments to final methods only. In the static case, the problem would also be solved by accessing the static method directly via the class type (and not polymorphically via the constructor). Both restrictions are severe but would be necessary to avoid unexpected runtime problems.

The following example shows a problem with breaking the encapsulation of a class.

```

class C {
    x: any = "";
    f(): void { this.g(this); }
    g(c: C): void { c.h(); }
    h(): void {}
}
class D extends C {

    @Override f(): void {
        this.g(this.x);
    }
    @Override g(c: any) {
        // do nothing, do not call h()
    }
}

var c = new C();
var d = new D();

var v5: {@This(C)function():void} = c.f;
var v6: {@This(C)function():void} = d.f;

v5.call(c)
v6.call(c)

```

In `D`, method `g` is overridden to accept more types as the original method defined in `C`. Calling this new method with receiver type `C` (as done in the last line) will cause problems, as in `D` not only `f` has been adapted but also `g`. Eventually, this would lead to a runtime error as well.

#### 5.2.3.4. Final Methods

By default, methods can be overridden. To prevent a method from being overridden, it must be annotated with `@Final`.

Of course, a method cannot be declared both abstract and final (cf. [Req-IDE-46]). Private methods are implicitly declared final. Because static methods can be overridden in subclasses (which is different to Java), they also can be marked as final.

Default methods in interfaces, cf. [Section 5.2.4, “Default Methods in Interfaces”](#), may also be declared `@Final`.

#### Example 5.7. Final Methods in Interfaces

If a method in an interface is provided with a body, it may be declared final. This will ensure that the given method's body will be in effect for all instances of the interface. Note that this means that;

- a class implementing that interface must not define a method with the same name and
- a class inheriting a method of that name cannot implement this interface.

The latter case is illustrated here:

```
interface I {
    @Final m(): void {}
}

class C1 {
    m(): void {}
}

// error at "I": "The method C1.m cannot override final method I.m."
class C2 extends C1 implements I {
}
```

### 5.2.3.5. Abstract Methods

A method can be declared without defining it, i.e. without providing a method body, and is then called an *abstract method*. Such methods must be declared with modifier `abstract` and have their property `abstract` set to true. Constraints for abstract methods are covered in [Req-IDE-46] (see [Section 5.1.3.4, “Abstract Classes”](#)).

In interfaces, methods are always abstract by default and they do not have to be marked as abstract. If a method in an interface provides a body, then this is the default implementation. See [Section 5.2.8.2, “Implementation of Members”](#) about how the default implementation may be mixed in the consumer.

### 5.2.3.6. Generic Methods

Methods of generic classes can, of course, refer to the type variables defined by type parameters of the generic class. These type variables are used similarly to predefined or declared types. Additionally, methods may be declared generic independently from their containing class. That is to say that type parameters (with type variables) can be defined for methods as well, just like for generic functions (see [Section 6.5.1, “Generic Functions”](#)).

#### Type variable names for generic methods

##### Requirement: IDE-55: Type variable names for generic methods (ver. 1)

For a given generic method `m` of a class `C`, the following constraint must hold:

$$\forall tp_m \in m.typePars, tp_C \in C.typePars : tp_m.name \neq tp_C.name$$

Since type variables can be used similarly to types in the scope of a generic class, a generic method may refer to a type variable of its containing class.

```
class C {
    <T> foo(p: T p): T { return p; }
}
```

If a generic type parameter is not used as a formal parameter type or the return type, a warning is generated unless the method overrides a member inherited from a super class or interface.

### 5.2.4. Default Methods in Interfaces

If a method declared in an interface defines a body, then this is the so-called *default implementation* and the method is called a *default method*. This will be mixed into an implementor of the interface if, and only if, neither the implementing class nor any of its direct or indirect superclasses already provides an implementation for this method; for details see [Section 5.2.8.2.1, “Member](#)

Consumption". Since the implementor is not known, some constraints exist for the body. I.e., no access to super is possible, cf. [Req-IDE-124].

In order to declare an interface to provide a default implementation in a definition file, annotation `@ProvidesDefaultImplementation` can be used, cf. [Req-IDE-167].

When a method in an interface is provided with a default implementation, it may even be declared `@Final`, see Section 5.2.3.4, "Final Methods".

#### 5.2.4.1. Asynchronous Methods

N4JS implements the async/await concept proposed for ECMAScript 7, which provides a more convenient and readable syntax for writing asynchronous code compared to using built-in type Promise directly. This concept can be applied to methods in exactly the same way as to declared functions. See Section 6.4.1, "Asynchronous Functions" and Section 6.4.1.3, "Asynchronous Arrow Functions" for details.

#### 5.2.5. Constructors

A constructor is a special function defined on a class which returns an instance of that class. The constructor looks like a normal method with name "constructor". The constructor can be defined explicitly or implicitly and every class has an (implicit) constructor.

For a given a class `C`, the constructor is available via two properties:

- `ownedCtor` the explicitly defined constructor (if any).
- `ctor` the explicit or implicit constructor.

If `C` is provided with an explicit constructor, we have `C.ctor = C.ownedCtor` and `C.ownedCtor ∈ C.ownedMembers`. Note that `C.ctor ≠ C.ownedMethods` in all cases.

The return type of the constructor of a class `C` is `C`. If `C` has type parameters  $T_1, \dots, T_n$ , then the return type is  $C < T_1, \dots, T_n >$ . The constructor is called with the operator. Since the return type of a constructor is implicitly defined by the class, it is to be omitted. By this definition, a constructor looks like the following:

```
class C {
    public constructor(s: string) {
        // init something
    }
}
```

Constructors define a variable execution environment and therefore provide access to the actual passed-in parameters through the implicit variable inside of their bodies (c.f. Section 6.1.4, "Arguments Object").

#### Defining and Calling Constructors

##### Requirement: IDE-56: Defining and Calling Constructors (ver. 1)

For a constructor `ctor` of a class `C`, the following conditions must hold:

1. `ctor` must neither be abstract nor static nor final and it must not be annotated with `@Override`.
2. If a class does not explicitly define a constructor then the constructor's signature of the superclass constructor is assumed.
3. If a class defines a constructor with formal parameters then this constructor has to be called explicitly in constructors defined in subclasses.
4. If a super constructor is called explicitly, this call must be the only expression of an expression statement which has to be the first statement of the body.
5. Constructors may appear in interfaces, but some restrictions apply:
  - a. constructors in interfaces must not have a body.
  - b. constructors in interfaces or their containing interface or one of its direct or indirect super interfaces must be annotated with `@CovariantConstructor`.
6. A constructor must not have an explicit return type declaration.
7. The implicit return type of a constructor is `this?`.
8. A constructor must not have any type parameters.

Properties of object literals may be called `constructor`. However they are not recognized as constructors in these cases.

#### Initialization of Final Fields in the Constructor

**Requirement: IDE-57: Initialization of Final Fields in the Constructor** (ver. 1)

- Required attributes must be initialized:

$$\forall a \in C.attr: a.required \Rightarrow \exists e \in r.elements: a.name = e.name$$

Note on syntax: ECMAScript 6 defines constructors similarly, [ECMA15a(p.S13.5)]. In ECMAScript 6 the super constructor is not called automatically as well.

The super literal used in order to call super methods is further described in [Section 8.2.1, “The super Keyword”](#).

#### 5.2.5.1. Structural This Type in Constructor

The use of a structural this reference as a formal parameter type is possible only in constructors. This parameter can be annotated with `@spec` which causes the compiler to generate initialization code.

Simply using `this` as a type in the constructor causes the constructor to require an object providing all public fields of the class for initialization purposes. The fields have to be set manually as shown in the following code snippet.

```
class A{
    public s: string;
    public constructor(src: ~this) {
        this.s = src.s;
    }
}
```

Remarks:

- The type of the formal parameter `~this` refers to the structural field type, see [Section 5.3, “Structural Typing”](#) for details on structural typing. It contains all public fields of the type.
- Subclasses may override the constructor and introduce additional parameters. They have to call the super constructor explicitly, however, providing a parameter with at least all required attributes of the superclass. Usually the type `this` is replaced with the actual subclass, but in the case of a `super()` call the `this` type of structural formal parameters is replaced with the `this` type of the superclass, hence only required fields of the superclass must be present.

As with other structural references, it is possible to add the structural reference with additional structural members, which can be used to initialize private fields which become not automatically part of the structural field type. For example:

```
class A{
    public s: string;
    private myPrivateNumber: number;
    public constructor(src: ~this with { x: number; }) {
        this.s = src.s;
        this.myPrivateNumber = src.x;
    }
}
```

Defining additional members may become a problem if a subclass defines public fields with the same name, as the `~this` type will contain these fields in the subclass. This is marked as an error in the subclass.

#### Names of additional members of structural this type in constructor

**Requirement: IDE-58: Names of additional members of structural this type in constructor** (ver. 1)

If the structural this type is used in a constructor of a class `C`, and if this structural reference contains an additional structural member `SM`, the following constraints must hold true:

- For any subclass `S` of `C`, with `S.ctor = C.ctor` (the subclass does not define its own constructor), `S` must not contain a public member with same name as `SM`:

$$S < C, S.ctor = C.ctor \Rightarrow \nexists M \in S.members: M.acc = public \wedge M.name = SM.name$$

- `C` itself must not contain a public member with same name as `SM`:

$$\nexists M \in C.members: M.acc = public \wedge M.name = SM.name$$

**Example 5.8. Field name conflicts with structural member name**

The situation described in [Req-IDF-58] is demonstrated in the following code fragment:

```
class A {
    private myPrivateNumber: number;
    public constructor(src: ~this with { x: number; }) {
        this.myPrivateNumber = src.x;
    }
}

class B extends A {
    public x: number; // will cause an error message
}
```

### 5.2.5.2. @Spec Constructor

The tedious process of copying the members of the parameter to the fields of the class can be automated via the `@spec` annotation if the argument has `~i~this` structural initializer field typing. More details about this typing can be found in Section 5.3.4, “Structural Read-only, Write-only and Initializer Field Typing”. This can be used as shown in the following listing:

```
class A {
    public field: string;
    public constructor(@spec spec: ~i~this) {}
}
let a = new A({field: 'hello'});
console.log(a.field); // prints: hello
```

The code for initializing the public field of `A` is automatically generated, thanks to the `@spec` annotation being given in the constructor.

#### @Spec Constructor

##### Requirement: IDE-59: @Spec Constructor (ver. 1)

1. Annotation `@spec` may only appear on a formal parameter of a constructor. Such a formal parameter is then called `@Spec parameter` or simply `spec parameter` and its owning constructor is referred to as a `@Spec constructor` or `spec constructor`. An argument to the spec parameter is called `spec object`.
2. Only a single formal parameter of a constructor may be annotated with `@spec`.
3. If a formal parameter is annotated with `@spec`, the parameter's type must be `~i~this` (i.e. a use-site structural initializer field type of `this`, see Section 5.3.4, “Structural Read-only, Write-only and Initializer Field Typing”).
4. Using the data provided in the spec object, i.e. in the argument to the spec parameter, a spec constructor will automatically initialize
  - a. all **owned** data fields and **owned** setters of the containing class, and
  - b. all data fields and setters from interfaces implemented by the containing class
 if and only if those members are also part of the spec parameter's structural initializer field type.
5. Fields explicitly added to the spec parameter, e.g. `@spec spec: ~i~this with {name:string}`, are used for initialization if a non-public field of the same name exists in the class, either as an owned member or from an implemented interface. The type of such an additional field must be a subtype of the declared type of the field being initialized:
 
$$\forall s \in ctor.fpar.structuralMembers, ctor.fpar.spec: \exists f \in ctor.owner.ownedFields \Rightarrow \Gamma \vdash s < : f$$
6. Even if the `@spec` annotation is used, the super constructor must be invoked explicitly (as usual).

It follows from no. 4 above that

1. non-public data fields and setters are never initialized (because they will never be part of the spec parameter's structural initializer field type),
2. properties provided in the spec object but not defined in the parameter's structural initializer field type, are *not* used for initialization, even if a (protected or private) field of the same name exists in the class,
3. data fields and setters inherited from a super class are never initialized by a spec constructor (instead, this will happen in the spec constructor of the super class).

The last of these implications will be detailed further at the end of the coming section.

### @Spec Constructors and Inheritance

Spec constructors are inherited by subclasses that do not have a constructor and, when creating instances of the subclass, will then require properties for writable public fields of the subclass in the spec object **and** include initialization code for them.

```
class A {
    public fa;
    public constructor(@Spec spec: ~i~this) {}
}
class B extends A {
    public fb;
}

const b = new B({fa: 'hello', fb: 'world'}); // requires & initializes fb too!
console.log(b.fa); // prints: hello
console.log(b.fb); // prints: world
```

Public writable fields from implemented interfaces are included as well, i.e. required as property in spec object **and** initialized by auto-generated code in the `@Spec` constructor:

```
interface I {
    public fi;
}
class B implements I {
    public fb;
    public constructor(@Spec spec: ~i~this) {}
}

const a = new B({fb: 'hello', fi: 'world'}); // requires & initializes fi too!
console.log(a.fb); // prints: hello
console.log(a.fi); // prints: world
```

When having a spec constructor in a class `B` that extends a super class `A` without an owned or inherited spec constructor, it should be noted that the `~i~this` type will require properties for public writable fields of `A`, but the initialization code automatically generated due to the `@Spec` annotation will **not** initialize those members. For public writable fields from an interface `I` implemented by `B`, however, both a property will be required by `~i~this` **and** initialization code will be generated in the `@Spec` constructor. This is illustrated in the following code example.

```
class A {
    public fa;
}
interface I {
    public fi;
}
class B extends A implements I {
    public fb;
    public constructor(@Spec spec: ~i~this) { // <- fa, fi, fb required in spec object
        // Constructor is responsible for initializing fa, fi, fb.
        // The @Spec annotation will generate initialization code
        // for fb and fi, but not for fa!
    }
}

let b = new B({
    fa: 'hello', // <- fa is required (removing it would be a compile error)
    fi: 'world',
    fb: '!!!'
});

console.log(b.fa); // undefined
console.log(b.fi); // world
console.log(b.fb); // !!!
```

The rationale for this different handling of fields from super classes and implemented interfaces is 1. fields from an implemented interface are not seen as inherited but rather implemented by implementing class, so from the `@Spec` annotation's perspective the field is a field of the implementing class, and 2. in case of a field inherited from a super class the correct way of initialization may depend on details of the super class and has to be taken care of by custom code in the constructor of the subclass (usually by invoking the non-`@Spec` constructor of the superclass with `super`).

### Special Use Cases

The following examples illustrate further details of other use cases of spec constructors.

**Example 5.9. Anonymous Interface in Constructor**

The base class `A` in the examples redefines the constructor already defined in `N4Object`. This is not generally necessary and is only used here to make the example legible.

```
class A {
    public s: string;
    public constructor(@Spec spec: ~i~this) {
        // initialization of s is automatically generated
    }
}
class B extends A {
    public t: string;
    private n: number;
    public constructor(spec: ~this with {n: number;}) {
        super(spec); // only inherited field s is set in super constructor
    }
}
```

**Example 5.10. Spec Object and Subclasses**

```
class A1 {
    public s: string;
    public n: number;
    public constructor(@Spec spec: ~i~this) {}
}
class B extends A1 {
    public constructor() {
        super({s:"Hello"}); // <-- error, n must be set in object literal
    }
}
class C extends A1 {
    public constructor() {
        super({s:"Hello"}); // <-- error, n must be set in object literal
        this.n = 10; // <-- this has no effect on the super constructor!
    }
}

class A2 {
    public s: string;
    public n: number?; // now n is optional!
    public constructor(@Spec spec: ~i~this) {}
}
class D extends A2 {
    public constructor() {
        super({s:"Hello"}); // and this is ok now!
        this.n = 10; // this explains why it is optional
    }
}

class A3 {
    public s: string;
    public n: number = 10; // now n is not required in ~this
    public constructor(@Spec spec: ~i~this) {}
}
class E extends A3 {
    public constructor() {
        super({s:"Hello"}); // and this is ok now!
    }
}
```

The last case (class `E`) demonstrates a special feature of the typing strategy modifier in combination with the `this` type, see Section 5.3, “Structural Typing” for details.

The constructor in class `B` contains an error because the super constructor expects all required attributes in `A1` to be set. The additional initialization of the required field `A1.n` as seen in `C` does not change that expectation. In this example, the field `n` should not have been defined as required in the first place.

Optional fields like `n?` in class `A2` or fields with default values like `n=10` in class `A3` are not required to be part of the `spec` object.

**Example 5.11. Superfluous Properties in @Spec Constructors**

Each non-public field has to be set in the constructor via the `with` to the parameter otherwise properties are *not* used to set non-public fields.

```

class C {
    public s: string;
    n: number;
    constructor(@Spec spec: ~i~this) {}
}

// n is ignored here
new C( { s: "Hello", n: 42 });

// but:
var ol = { s: "Hello", n: 42 };
// "ol may be used elsewhere, we cannot issue warning here" at "ol"
new C(ol);

// of course this is true for all superfluous properties
// weird is not used in constructor
new C( { s: "Hello", weird: true } );

```

<b>Restriction when initializing interface fields via @Spec constructor</b>	In most cases, interface definitions in <code>n4jsd</code> files simply declare functions and fields that are supposed to be provided by the runtime environment. As a result, there are restrictions as to whether fields of interfaces defined in <code>n4jsd</code> files can be initialized via <code>@spec</code> constructors or not. In particular, fields of an interface declared in a <code>n4jsd</code> file cannot be initialized via <code>@Spec</code> constructor if the interface
---	---

1. is a built-in or
2. does not have an `@N4JS` annotation

The following example illustrates this restriction.

#### Example 5.12. Interface fields that cannot be initialized via @Spec constructors

##### Inf.n4jsd

```

export external interface I {
    public m: string;
}

@N4JS
export external interface J {
    public n: string;
}

```

##### Test.n4js

```

import { I } from "Inf";
// I is an external interface WITHOUT @N4JS annotation
class C implements I {
    constructor(@Spec spec:~i~this) {}
}

// J is an external interface with @N4JS annotation
class D implements J {
    constructor(@Spec spec:~i~this) {}
}

// XPECT warnings --> "m is a property of built-in / provided by runtime / external without @N4JS annotation interface I
// and can not be initialized in Spec constructor." at "m"
let c:C = new C({m: "Hello"});

// XPECT nowarnings
let d:D = new D({n: "Bye"});

console.log(c.m)
console.log(d.n)

/* XPECT output ---
<==
stdout:
undefined
Bye
stderr:
==>
--- */

```

In this example, the interface `I` is defined in the `Inf.n4jsd` file without the `@N4JS` annotation. As a result, its field `m` cannot be initialized via the `@Spec` constructor and hence the output of `console.log(c.m)` is `undefined`. On the other hand, since the

interface `J` is declared with the annotation `@N4JS`, it is possible to initialize its field `n` in the `@Spec` constructor. That's why the result of `console.log(d.n)` is `Bye`.

### 5.2.5.3. Callable Constructors

### 5.2.5.4. Covariant Constructors

Usually, the constructor of a subclass need not be override compatible with the constructor of its super class. By way of annotation `@CovariantConstructor` it is possible to change this default behavior and enforce all subclasses to have constructors with override compatible signatures. A subclass can achieve this by either inheriting the constructor from the super class (which is usually override compatible, with the special case of `@Spec` constructors) or by defining a new constructor with a signature compatible to the inherited constructor. The same rules as for method overriding apply.

The `@CovariantConstructor` annotation may be applied to the constructor, the containing classifier, or both. It can also be used for interfaces; in fact, constructors are allowed in interfaces only if they themselves or the interface is annotated with `@CovariantConstructor` (see [Req-IDE-60]).

#### Covariant Constructor

##### Definition: Covariant Constructor

A classifier `C` is said to *have a covariant constructor* if and only if one of the following applies:

1. `C` has a direct super class `C'` and `C'` is annotated with `@CovariantConstructor` or `C'` has a constructor annotated with `@CovariantConstructor`.
2. `C` has a directly implemented interface `I` and `I` is annotated with `@CovariantConstructor` or `I` has a constructor annotated with `@CovariantConstructor`.
3. `C` has a direct super class or directly implemented interface that *has a covariant constructor* (as defined here).

Note that `C` does not need to have an owned(!) constructor; also a constructor inherited from a super class can be declared covariant.

The following rules apply to covariant constructors.

#### Covariant Constructors

##### Requirement: IDE-60: Covariant Constructors (ver. 1)

. Annotation `@CovariantConstructor` may only be applied to classes, interfaces, and constructors. Annotating a constructor with this annotation, or its containing classifier, or both have all the same effect. . Given a class `C` with an owned constructor `ctor` and a super class `Sup` that has a covariant constructor (owned or inherited, see Section 5.2.5.4, "Covariant Constructors"), then `Sup.constructor` must be accessible from `C`, .. `ctor` must be override compatible with `Sup.constructor`:

+ `overrideCompatible(ctor, Sup.constructor)`

+ This constraint corresponds to [Req-IDE-72] except for the `Override` annotation which is not required here. . Given a classifier `C` implementing interface `I` and `I` has a covariant constructor (owned or inherited, see Section 5.2.5.4, "Covariant Constructors"), we require .. `I.constructor` must be accessible from `C`, .. an implementation-compatible constructor `ctor` must be defined in `C` with

+ `overrideCompatible(ctor, I.constructor)`

+ This constraint corresponds to [Req-IDE-74] except for the `@Override` annotation, which is not required, here. . Given a classifier `C` without an owned constructor and an extended class or interface `Sup` that has a covariant constructor (owned or inherited, see Section 5.2.5.4, "Covariant Constructors"), we require the inherited constructor `ctor` of `C` within the context of `C` to be override compatible to itself in the context of `Sup`. Using notation `m[T]` to denote that a member `M` is to be treated as defined in container type `T`, which means the this-binding is set to `T`, we can write:

+ `overrideCompatible(ctor[C], ctor[Sup])`

+ This constraint does not correspond to any of the constraints for the redefinition of ordinary members.

The following example demonstrates a use case for covariant constructors. It shows a small class hierarchy using covariant constructors, `cls1` and `cls2`, together with a helper function `createAnother` that creates and returns a new instance of the same type as its argument `value`.

**Example 5.13. Covariant Constructors**

```

class A {}
class B extends A {}

@CovariantConstructor
class Cls {
    constructor(p: B) {}
}
class Cls2 extends Cls {
    constructor(p: A) { // it's legal to generalize the type of parameter 'p'
        super(null);
    }
}

function <T extends Cls> createAnother(value: T, p: B): T {
    let ctor = value.constructor;
    return new ctor(p);
}

let x = new Cls2(new A());
let y: Cls2;

y = createAnother(x, new B());

```

In the code of [Example 5.13, “Covariant Constructors”](#), we would get an error if we changed the type of parameter `p` in the constructor of `Cls2` to some other type that is not a super type of `B`, i.e. the type of the corresponding parameter of `Cls`'s constructor. If we removed the `@CovariantConstructor` annotation on `Cls`, we would get an error in the new expression inside function `createAnother`.

The next example illustrates how to use `@CovariantConstructor` with interfaces and shows a behavior that might be surprising at first sight.

**Example 5.14. Covariant Constructors in Interfaces**

```

@CovariantConstructor
interface I {
    constructor(p: number)
}

class C implements I {
    // no constructor required!
}

class D extends C {
    // EXPECT errors --> "Signature of constructor of class D does not conform to overridden constructor of class N4Object:
    // {function(number)} is not a subtype of {function()}" at "constructor"
    constructor(p: number) {}
}

```

Interface `I` declares a covariant constructor expecting a single parameter of type `number`. Even though class `C` implements `I`, it does not need to define an owned constructor with such a parameter. According to [\[Req-IDE-60\]](#), it is enough for `C` to have a constructor, either owned or inherited, that is override compatible with the one declared by `I`. Class `C` inherits the default constructor from `N4Object`, which does not have any arguments and is thus override compatible to `I`'s constructor.

In addition, subclasses are now required to have constructors which are override compatible with the constructor of class `C`, i.e. the one inherited from `N4Object`. [Example 5.14, “Covariant Constructors in Interfaces”](#) shows that this is violated even when repeating the exact same constructor signature from interface `I`, because that constructor now appears on the other side of the subtype test during checking override compatibility.

## 5.2.6. Data Fields

A data field is a simple property of a class. There must be no getter or setter defined with the same name as the data field. In ECMAScript 6, a class has no explicit data fields. It is possible, however, to implicitly define a data field by simply assigning a value to a variable of the `this` element (e.g. `this.x = 10` implicitly defines a field `x`). Data fields in N4JS are similar to these implicit fields in ECMAScript 6 except that they are defined explicitly in order to simplify validation and user assistance.

### 5.2.6.1. Syntax

```

N4FieldDeclaration <Yield>:
    {N4FieldDeclaration}

```

```

FieldDeclarationImpl<Yield>
;

fragment FieldDeclarationImpl <Yield>*:
  (declaredModifiers=N4Modifier)* BogusTypeRefFragment?
  declaredName=LiteralOrComputedPropertyName<Yield>
  (declaredOptional?= '?' )?
  ColonSepTypeRef?
  ('=' expression=Expression<In=true,Yield>)?
  Semi
;

```

### 5.2.6.2. Properties

Fields have the following properties which can be explicitly defined:

<code>declaredOptional</code>	Tells whether the accessor was declared optional.
<code>typeRef</code>	Type of the field; default value is <code>Any</code> .
<code>expr</code>	Initializer expression, i.e. sets default value.
<code>static</code>	Boolean flag set to true if field is a static field.
<code>const</code>	Boolean flag set to true if field cannot be changed. Note that const fields are automatically static. Const fields need an initializer. Also see <a href="#">Section 5.2.6.3, "Assignment Modifiers"</a> .



`const` is not the (reversed) value of the property descriptor `writable` as the latter is checked at runtime while `const` may or may not be checked at runtime.

The following pseudo properties are defined via annotations for setting the values of the property descriptor:

<code>enumerable</code>	Boolean flag reflecting the property descriptor <code>enumerable</code> , set via annotation <code>@Enumerable(true false)</code> . The default value is <code>true</code> . <sup>6</sup>
<code>declaredWriteable</code>	Boolean flag reflecting the property descriptor <code>writeable</code> , set via annotation <code>@Writeable(true false)</code> . The default value is <code>true</code> . <sup>7</sup>
<code>final</code>	Boolean flag making the field read-only, and it must be set in the constructor. Also see <a href="#">Section 5.2.6.3, "Assignment Modifiers"</a> .

#### Derived values for fields

<code>readable</code>	Always true for fields.
<code>abstract</code>	Always false for fields.
<code>writable</code>	Set to false if field is declared <code>const</code> or <code>final</code> . In the latter case, it may be set in the constructor (cf. <a href="#">Section 5.2.6.3, "Assignment Modifiers"</a> ).

### 5.2.6.2.1. Semantics

#### Attributes

##### Requirement: IDE-61: Attributes (ver. 1)

For any attribute `a` if a class `c`, the following constraints must hold:

1. A required data field must not define an initializer:  
 $arequired \Rightarrow a.init = null$
2. There must be no other member with the same name of a data field `f`. In particular, there must be no getter or setter defined with the same name:  
 $\forall m \in f.owner.members: m \neq f \Rightarrow m.name \neq f.name$

If a subclass should set a different default value, this has to be done in the constructor of the subclass.

For the relation of data fields and field accessors in the context of extending classes or implementing interfaces see [Section 5.2.8, "Redefinition of Members"](#).

<sup>6</sup> version 4.0

<sup>7</sup> This cannot be done w/o `null/undefined` analysis

### 5.2.6.2.2. Type Inference

The type of a field is the type of its declaration:

$$\frac{\Gamma \vdash f : \Gamma \vdash d}{\Gamma \vdash d}$$

The type of a field declaration is either the declared type or the inferred type of the initializer expression:

$$\frac{d.declaredType \neq \text{null} \quad T = d.declaredType}{\Gamma \vdash d:T}$$

$$\frac{d.declaredType = \text{null} \quad d.expression \neq \text{null}}{\Gamma \vdash d:T}$$

$$\frac{E = \Gamma \vdash d.expression \quad E \notin \{\text{null}, \text{undefined}\} \quad T = E}{\Gamma \vdash d:\text{any}}$$

If the type contains type variables they are substituted according to type parameters which are provided by the reference:

$$\frac{\Gamma \vdash tfield.typeRef : T}{\Gamma \vdash \text{TField } tfield : T}$$

### 5.2.6.3. Assignment Modifiers

Assignment of data fields can be modified by the assignment modifiers `const` (similar to constant variable declarations, see [Section 9.2.2, “Const”](#)) and `@Final`.

#### Const Data Fields

##### **Requirement: IDE-62: Const Data Fields (ver. 1)**

For a data field `f` marked as `const`, the following constraints must hold:

1. An initializer expression must be provided in the declaration (except in n4jsd files):  
 $f.expr \neq \text{null}$
2. A constant data field is implicitly static and must be accessed only via the classifier type. It is not possible, therefore, to use the `this` keyword in the initializer expression of a constant field:  
 $\nexists sub \in f.expr^* : sub = "this"$
3. A constant data field must not be annotated with `@Final`:  
 $f.const \rightarrow \neg f.final$
4. Constant data fields are not writeable (cf. [\[Req-IDE-68\]](#)):  
 $f.const \rightarrow \neg f.writeable$

#### Final Data Fields

##### **Requirement: IDE-63: Final Data Fields (ver. 1)**

For a data field `f` marked as `@Final`, the following constraints must hold:

1. A final data field must not be modified with `const` or `static`:  
 $f.final \rightarrow \neg f.const \wedge \neg f.declaredStatic$
2. A final data field is not writeable:  
 $f.final \rightarrow \neg f.writeable$   
A final field may, however, be set in the constructor. See [\[Req-IDE-68\]](#) for details.
3. A final data field must be either initialized by an initializer expression or in the constructor. If the field is initialized in the constructor, this may be done either explicitly or via a spec style constructor.

$$f.expr \neq \text{null}$$

$$\vee (\exists assignExp : assignExpr \text{ containingFunction} = f.owner.constructor \wedge assignExpr.left.target = "this" \wedge bind(assignExpr.left.property, f))$$

$$\vee (f.public \wedge \exists fpar \in f.owner.constructor.fpars : fpar.spec \wedge \exists sm \in structuralMembers : sm.name = f.name)$$

## 5.2.6.4. Field Accessors (Getter/Setter)

Instead of a simple data field, a field can be defined by means of the getter and setter accessor methods. These accessor methods are similar to the accuser methods in object literals:

### 5.2.6.4.1. Syntax

```
N4GetterDeclaration <Yield>:
  => ({N4GetterDeclaration}
    (declaredModifiers+=N4Modifier)*
    GetterHeader<Yield>)
    (body=Block<Yield>)? ';'?
;

fragment GetterHeader <Yield>*:
  BogusTypeRefFragment? 'get' -> declaredName=LiteralOrComputedPropertyName<Yield>
  (declaredOptional?=?)?
  '(' ')'
  ColonSepTypeRef?
;

N4SetterDeclaration <Yield>:
  => ({N4SetterDeclaration}
    (declaredModifiers+=N4Modifier)*
    'set'
    ->declaredName=LiteralOrComputedPropertyName <Yield>
  )
  (declaredOptional?=?)?
  '(' fpar=FormalParameter<Yield> ')' (body=Block<Yield>)? ';'?
;
```

Notes with regard to syntax: Although ECMAScript 6 does not define fields in classes, it defines getter and setter methods similarly (cf. [ECMA15a(p.S13.3, p.p.209)]).

#### Example 5.15. Getter and Setter

The getter and setter implementations usually reference data fields internally. These are to be declared explicitly (although ECMAScript allows creating fields on the fly on their first usage). The following example demonstrates a typical usage of getter and setter in combination with a data field. The getter lazily initializes the field on demand. The setter performs some notification.

#### Getter Setter

```
class A {}

class C {
  private _data: A = null;

  public get data(): A {
    if (this._data==null) {
      this._data = new A();
    }
    return this._data;
  }

  public set data(data: A) {
    this._data = data;
    this.notifyListeners();
  }

  notifyListeners(): void {
    // ...
  }
}
```

## 5.2.6.4.2. Properties

Properties for field accessors:

<b>declaredOptional</b>	Tells whether the accessor was declared optional.
<b>readable</b>	Derived value: true for getters and false for setters.
<b>writable</b>	Derived value: false for getters and true for setters.

### 5.2.6.4.3. Semantics

There must be no field or method with the same name as a field accessor (follows from [Req-IDE-52]). In addition, the following constraints must hold:

#### Field Accessors

##### Requirement: IDE-64: Field Accessors (ver. 1)

- The return type of a getter must not be `void`.
- The type of the parameter of a setter must not be `void`.
- If a getter `g` is defined or consumed (from an interface) or merged-in (via static polyfill) in a class `C` and a setter `s` with `s.name = g.name & s.static = g.static` is inherited by `C` from one of its super classes, then `C` must define a setter `s'` with `s'.name = g.name & s'.static = g.static`<sup>8</sup>.
- A setter must have exactly one formal parameter, i.e. variadic or default modifiers are not allowed.

The same applies to setters, accordingly.

- [Req-IDE-72], [Req-IDE-73], and [Req-IDE-74] apply to field accessors accordingly (getter / setter overriding).



A getter and setter with the same name need not have the same type, i.e. the getter's return type need not be the same as a subtype of the type of the setter's parameter (the types can be completely unrelated).<sup>9</sup>

Getters and setters – like functions – define a variable execution environment and therefore provide access to the actual passed-in parameters through the implicit `arguments` variable inside of their bodies (c.f. Section 6.1.4, “Arguments Object”).

### 5.2.6.5. Optional Fields

Data fields and field accessors of a classifier `C` can be declared optional, meaning that a structural subtype of `C` need not provide this field, but if it does, the field must be of correct type. However, to ensure overall type safety, the scope of application of this optionality is limited to a small number of specific use cases, as described in the following.

#### 5.2.6.5.1. Syntax

To denote a data field or accessor as optional, a question mark is placed right after the name:

##### Syntax of optional fields

```
class C {
    public field?: string;

    public get getter?(): number {
        return 42;
    }
    public set setter?(value: number) {}
}
```

The detailed grammar is given in the sections for data fields, cf. Section 5.2.6.1, “Syntax”, and field accessors, cf. Section 5.2.6.4.1, “Syntax”.

#### 5.2.6.5.2. Semantics

It is important to note that the optionality of a field is, by default and in most cases, ignored and has an effect only in certain special cases.

The effect of a field being optional is defined by the following requirement.

#### Optional Fields

<sup>8</sup> This is required, because in Javascript a getter shadows a corresponding setter defined further up in the prototype chain; likewise a setter shadows a corresponding getter.

<sup>9</sup> Thus, the type of one accessor is not used to infer the type of the other one. E.g., from `set x(string s)`, we cannot infer `get x()` to return `string` — instead, the getter is inferred to return `any`.

**Requirement: IDE-240500: Optional Fields (ver. 1)**

By default, a data field, getter, or setter that is declared optional is handled in the exact same way as if no optionality were involved (i.e. by default, optionality is ignored).

Optionality has an effect only in case of structural subtype checks  $L < :R$  in which the left-hand side is one of the following:

1. an object literal.
2. a new expression.
3. an instance of a final class, i.e. the type of the value on left-hand side must be nominal and refer to a final class.
4. a reference to a const variable if its initializer expression is one of the following:
  - a. an object literal.
  - b. a new expression.
  - c. an instance of a final class (as explained above).
  - d. a ternary expression

and then

- in cases 1 and 4a, **both** fields and accessors (getters and setters) are optional. That means, an optional data field, getter, or setter of  $R$  needs not be present in  $L$ .
- in cases 2, 3, 4b, and 4c, **only getters** are optional, setters are not optional. That means, an optional getter of  $R$  needs not be present in  $L$  and an optional field of  $R$  requires only a setter in  $L$ . Note that these cases are more restricted than the cases 1 and 4a.

Moreover, optionality has an effect in case of ternary expression  $L < :R$  in which the left-hand side is a ternary expression, e.g. `1 = b? trueExpr : falseExpr` whose `trueExpr` or `falseExpr` possibly recursively contains an expression of the kind mentioned above. In this case, the optionality effect is the more restricted optionality of `trueExpr` and `falseExpr`.

If, according to these rules, a data field / getter / setter of  $R$  need not be present in  $L$  but a member with the same name and access is actually present in  $L$ , that member in  $L$  must be a data field / getter / setter of the same type / a subtype / a super type, respectively. In other words, if a not actually required member is present in the subtype, ordinary rules for member compatibility apply as if no optionality were involved (cf. general subtyping rules for structural types).

In other words, in object literals (cases 1 and 4a) neither optional getters, optional setters, nor optional data fields are required. However, in case of new expressions and instances of final classes (cases 2, 3, 4b, 4c) only optional getters are not required in a subtype; optional setters are required as normal (i.e. optionality ignored) and optional data fields require at least a setter.

The following table summarizes the most common cases and shows how this relates to the different forms of structural typing.

**Table 5.2. Optional Fields**

	$\Delta$					Case	Comment
	$\sim$	$\sim\sim$	$\sim w \sim$	$\sim r \sim$	$\sim i \sim$		
	may have setter			never has setter			
<code>let x: #C = {};</code>	✓	✓	✓	✓	✓	1	nothing mandatory
<code>let x: #C = new D0();</code>				✓	✓	2	setters mandatory
<code>let x: #C = new DG();</code>				✓	✓	2	setters mandatory
<code>let x: #C = new DS();</code>	✓	✓	✓	✓	✓	2	setters mandatory
<code>let x: #C = fooD0();</code>						none	D0 not final
<code>let x: #C = foosF0();</code>						none	fooSF0() not nominal
<code>let x: #C = fooF0();</code>				✓	✓	3	setters mandatory

In the table, a "#" means that the particular example is valid; in all other cases an error would be shown in N4JS source code. Here are the classes and functions used in the above table:

**Classes and functions used in table**

```

class C {
    public field?: string;
}

class D0 {}

class DG {
    public get field(): string { return "hello"; }
}

class DS {
    public set field(value: string) {}
}

@Final class F0 {}

function fooD0(): D0 { return new D0(); }
function fooF0(): ~F0 { return new F0(); }
function fooF0(): F0 { return new F0(); }

```

It follows from the above definitions in Requirements [Req-IDE-240500] that cases 4a and 4b are not transitive across a chain of several `const` variables, whereas case 4c is transitive. For example:

#### Transitivity of the use cases of optional fields

```

class C {
    public get getter?(): string { return null; }
}
class D {}
@Final class F {}

let c: ~C;

// no transitivity via several const variables in use case "object literal":

const o1 = {};
const o2 = o1;

// XPECT errors --> "--Object is not a structural subtype of ~C: missing getter getter." at "o2"
c = o2;

// no transitivity via several const variables in use case "new expression":

const new1 = new D();
const new2 = new1;

// XPECT errors --> "D is not a structural subtype of ~C: missing getter getter." at "new2"
c = new2;

// BUT: we do have transitivity via several const variables in use case "final nominal type":

const finalNominal1 = new F();
const finalNominal2 = finalNominal1;

// XPECT noerrors -->
c = finalNominal1;
// XPECT noerrors --> "transitivity applies in this case"
c = finalNominal2;

```

The following example demonstrates how optionality behaves in ternary expressions.

#### Optional fields in ternary expressions

```

interface ~I {
    public m?: int;
}

class ~C {}

@Final class F {}

let b: boolean;
const cc: C = {};
let f1 = new F();
let f2: ~F = {};

// True expression is a const object literal, so both fields and accessors in I are optional.

```

```
// False expression is a new expression, so only getters in I are optionals.
// As a result, only getters in I are optional.
// XPECT errors --> "C is not a structural subtype of I: missing field m." at "b? cc : new C()"
var tel: I = b? cc : new C()

// No errors because both true and false expressions are object literal constants and hence
// Both fields and accessors in I are optional.
// XPECT noerrors
var te2: I = b? cc : {}
```

### 5.2.6.5.3. Background

The following example illustrates why optionality of fields has to be restricted to the few special cases defined above (i.e. object literals, new expressions, etc.).

#### Problem 1 of optional fields

```
class C {
    public field?: string = "hello";
}

class D {}
class DD extends D {
    public field: number = 42;
}

let c: ~C;
let d: D;

d = new DD();

c = d; // without the restrictive semantics of optional fields, this assignment would be allowed (but shows compile-time error in N4JS)

console.log(c.field); // prints 42 even though the type is string
c.field.charAt(0); // exception at runtime: c.field.charAt is not a function
```

In the last line of the above example, `c.field` is actually `42` but the type system claims it is of type `string` and thus allows accessing member `charAt` of type `string` which is undefined at runtime the actual value `42`.

The next example shows why cases 2 and 3 (i.e. new expressions and instances of final classes) have to be handled in a more restrictive manner than case 1 (i.e. object literals).

#### Problem 2 of optional fields

```
class C {
    public field?: string;
}

class D {}

let c: ~C;

c = new D(); // error: new expression but D is missing setter

c.field = "hello";
```

In the previous code, if `c = new D()` were allowed, we would add a new property `field` to the instance of class `D` in the last line, which N4JS aims to avoid in general, unless unsafe language features such as dynamic types are being employed.

### 5.2.7. Static Members

Static data fields, field accessors and methods are quite similar to instance members, however they are not members of instances of the type but the type itself. They are defined similarly to instance members except that they are specified with the modifier `static`. Since they are members of the type, the `this` keyword is not bound to instances of the class, but again to the type itself. This is similar as in ECMAScript 6 ([ECMA15a(p.14.5.15)]). Since static members are not instance but type members, it is even possible that a static member has the same name as an instance member.

Note that static members are not only allowed in classes but also in interfaces, but there are important differences (for example, no inheritance of static members of interfaces, cf. Section [Section 5.2.7.3, “Static Members of Interfaces”](#)).

Static member not abstract

**Requirement: IDE-65:** [Static member not abstract](#) (ver. 1)

For a static field accessor or method `s`, the following constraint must hold:

- $s \text{static} \Leftrightarrow \neg s.\text{abstract}$

Like instance methods, static methods of classes are inherited by subclasses and it is possible to override static methods in subclasses. The very same override constraints are valid in this case as well.

### 5.2.7.1. Access From and To Static Members

#### Accessing Static Members

##### Requirement: IDE-66: Accessing Static Members (ver. 1)

Let `m` be a static member of class `c`. Except for write-access to fields, which will be explained later, you can access `m` via:

1. The class declaration instance, i.e. the classifier or constructor type, `constructor{C}`, i.e. `C.m`
2. The class declaration instance of a subtype, i.e. the classifier or constructor type, i.e. `D.m`, if `D` is a subclass of `c`.
3. `v.m`, if `v` is a variable of type `c` (i.e. classifier type as defined in [Section 4.11, “Constructor and Classifier Type”](#)) or a subtype thereof.
4. `this.m` inside the body of any static method declared in `c` or any sub-class of `c`.
5. Via a type variable `T` which upper bound is a subclass of `c` e.g., `function <T extends C> f(){T.m}`

#### Static Member Access

##### Requirement: IDE-67: Static Member Access (ver. 1)

It is not possible to access instance members from static members. This is true in particular for type variables defined by a generic classifier.

#### Write-access to static data fields and static setter

##### Requirement: IDE-68: Write-access to static data fields and static setter (ver. 1)

For static data fields and static setter `f` the following constraint must hold:

- For every assign expression `assignExpr` with `f.static ∧ assignExpr.left = T.f ⇒ T = f.owner`.
- For every writing unary expression `u` with `u.op ∈ {++, --} ∧ f.static ∧ u.expression = T.f ⇒ T = f.owner`.

In the special case of `m` being a static data field, write-access is only possible via the defining type name `c.m`. In the list above, only the first line can be used when assigning values to a field. Note that this only applies to fields and set-accessors.<sup>10</sup>

It is even possible to call a static field accessor or method of a class using dynamic polymorphism, as demonstrated in the following example:

#### Example 5.16. Static members of classes, inheritance and polymorphism

```
class A {
    static m(): void { console.log('A#m'); }

    static foo(): void { console.log('A#foo'); }

    static bar(): void {
        this.foo();
    }
}

class B extends A {
    @Override
    static foo(): void { console.log('B#foo'); }
```

<sup>10</sup> The technical reason for this rule is the way properties are stored in JavaScript. Take for an example subclass-write access: `class C { static f="a"; } with class D extends C { } . Now the data field f on C can also be queried using D (var q=D.f; ) but writing (D.f="b"; ) would introduce a newly created property f on class D, which differs from the one defined on C. It would do this without explicitly overriding the inherited property. Subsequent reads to D.f would route to this ‘accidentally’ introduced property. Such a behavior would not be expected and therefore has been disallowed. Note that this write restriction applies to data-fields and to field setters.`

```

}

A.m(); // will print "A#m"
B.m(); // will print "A#m" (m is inherited by B)

var t: type{A} = A;
t.foo(); // will print "A#foo"
t = B;
t.foo(); // will print "B#foo"

// using 'this':

A.bar(); // will print "A#foo"
B.bar(); // will print "B#foo"

```

This is quite different from Java where static methods are not inherited and references to static methods are statically bound at compile time depending on the declared type of the receiver (and not its value):

#### Example 5.17. Static members in Java

```

// !!! JAVA CODE !!!
public class C {

    static void m() { System.out.println("C#m"); }

    public static void main(String[] args) {
        final C c = null;
        c.m(); // will print "C#m" (no NullPointerException at runtime)
    }
}

```

### 5.2.7.2. Generic static methods

It is not possible to refer to type variables of a generic class, as these type variables are never bound to any concrete types. A static method can, however, be declared generic. Generic static methods are defined similarly to generic instance methods. Since they cannot refer to type variables of a generic class, the constraint to avoid type variables with equal names (see [Req-IDE-55]) does not need to hold for generic static methods.

### 5.2.7.3. Static Members of Interfaces

Data fields, field accessors and methods of interfaces may be declared static. A few restrictions apply:

#### Static Members of Interfaces

##### Requirement: IDE-69: Static Members of Interfaces (ver. 1)

1. Static members of interfaces may only be accessed directly via the containing interface's type name (this means, of the four ways of accessing static members of classes defined in [Req-IDE-66] above, only the first one applies to static members of interfaces).
2. The `this` literal may not be used in static methods or field accessors of interfaces and it may not be used in the initializer expression of static fields of interfaces. See [Req-IDE-173].
3. The `super` literal may not be used in static methods or field accessors of interfaces (in fact, it may not be used in interfaces at all, cf. [Req-IDE-123]).

Note that the `this` type as a return type for methods is only allowed for instance methods and as an argument type only in constructors (structurally typed). There is no need to disallow these cases for static interface methods in the constraints above.

In general, static members may not be abstract, cf. [Req-IDE-46], which applies here as well. Static methods and field accessors of interfaces, therefore, always have to provide a body.

Static members of interfaces are much more restricted than those of classes. Compare the following example to Example 4.16, “Static Polymorphism” for classes above:

#### Example 5.18. Static members of interfaces

```

interface I {
    static m(): void { console.log('I#m'); }
}

```

```

interface J extends I {}

I.m(); // prints "I#m"
J.m(); // ERROR! (m is not inherited by J)

var ti: type{I} = I;
ti.m(); // ERROR! (access to m only allowed directly via type name I)
ti = J;
ti.m(); // ERROR! (access to m only allowed directly via type name I)

```

The last line in is the reason why access to static members has to be restricted to direct access via the type name of the containing interfaces.

### 5.2.8. Redefinition of Members

Members defined in classes or interfaces can be redefined by means of being overridden or implemented in subclasses, subinterfaces, or implementing classes. Fields and methods with default implementation defined in interfaces can be consumed by the implementor, but certain restrictions apply.

#### Override Compatible

##### Requirement: IDE-70: Override Compatible (ver. 1)

A member  $M$  is *override compatible* to a member  $S$  if and only if the following constraints hold:

1. The name and static modifiers are equal:

$$M.name = S.name \wedge M.static = S.static$$

2. The metatypes are compatible:

$$\begin{aligned}\mu(S) = \text{Method} &\rightarrow \mu(M) = \text{Method} \\ \mu(S) = \text{Field} &\rightarrow \mu(M) \in \{\text{Field, Getter, Setter}\} \\ \mu(S) = \text{Getter} &\rightarrow \mu(M) \in \{\text{Field, Getter}\} \\ \mu(S) = \text{Setter} &\rightarrow \mu(M) \in \{\text{Field, Setter}\}\end{aligned}$$

3. The overridden member must not be declared final:

$$\neg S.final$$

4. Overridden member declared const can only be overridden (redefined) by const members:

$$S.const \Rightarrow M.const$$

5. It is not possible to override a non-final / non-const field or a setter with a final / const field:

$$(\mu(S) = \text{Field} \wedge \neg (S.final \vee S.const)) \vee \mu(S) = \text{Setter} \Rightarrow \neg (\mu(M) = \text{Field} \wedge (M.final \vee M.const))$$

6. It is not possible to override a non-abstract member with an abstract one:

$$\neg M.abstract \vee S.abstract$$

7. The types are compatible:

$$\begin{aligned}(\mu(M) \in \{\text{Method, Getter, Field}\} \wedge \mu(S) \neq \text{Setter}) &\rightarrow \Gamma \vdash M < : S \\ (\mu(M) \in \{\text{Setter}\} \wedge \mu(S) \neq \text{Getter} \wedge \neg S.const) &\rightarrow \Gamma \vdash S < : M\end{aligned}$$

8. The access modifier is compatible:

$$M.access \geq S.access$$

We define a relation  $overrideCompatible(M, S)$  accordingly.

Members overriding or implementing other members must be declared as override. If a member does not override another, however, it must not be declared as override.

#### Non-Override Declaration

##### Requirement: IDE-71: Non-Override Declaration (ver. 1)

If and only if a member  $M$  of a class  $C$  (extending a class  $S$  and interfaces  $I_j$ ) does not override or implement another member, then it must not be declared as override. That is the following constraint must hold:

$$\begin{aligned}&\neg M.override \\ &\wedge\end{aligned}$$

$$\begin{aligned} \#M' \in C.super.members \cup \bigcup_{i=1}^n I_i.members: \\ M'.name = M.name \wedge M'.static = M.static \\ \wedge M'.acc > \text{private} \end{aligned}$$

### 5.2.8.1. Overriding of Members

In general, the N4JS platform supports overriding members by redefining them in sub-classes. This definition allows for overriding of static methods, but it does not apply to constructors because  $C.ctor \notin C.ownedMethods$ .

#### Overriding Members

##### Requirement: IDE-72: Overriding Members (ver. 1)

Given a class  $C$  and a superclass  $Sup$ . If for an instance or static member  $M$  defined in  $C$  a member  $S$  exists with null then we call  $M$  the overriding member and  $S$  the overridden member. In that case the following constraints must hold:

1.  $S$  must be accessible from  $C$
2.  $M$  must be override compatible with  $S$ :  

$$\text{overrideCompatible}(M, S)$$
3. If  $S$  is a field and  $M$  is an accessor, then an additional accessor  $M'$  must exist so that  $M, M'$  are an accessor pair for  $S$ :  

$$\begin{aligned} \mu(S) = \text{Field} \wedge \mu(M) = \text{Accessor} \\ \rightarrow \exists M' \in C.member: \\ \text{overrideCompatible}(M', S) \wedge \{\mu(M), \mu(M')\} = \text{Getter, Setter} \end{aligned}$$
4.  $M$  must be declared as override:  

$$M.override$$

Remarks:

- An overridden method, getter, or setter may be called via `super`. Note that this is not possible for fields.
- There is no 'hiding' of fields as in Java, instead there is field overriding.
- It is not possible to override a field with a consumed getter and an overridden setter, because the getter is not consumed if there exists a field in a superclass. In this case, the consuming and extending class needs to define the accessor pair explicitly. The same is true for other combinations of accessors and fields.
- Overriding a field usually makes sense if the visibility of the field is to be increased.

### 5.2.8.2. Implementation of Members

#### Interface and Class Member Sets

##### Definition: Interface and Class Member Sets

For the following constraints, we define two helper sets  $M_C$  and  $M_I$  as follows:

Given a  $C$ , and interface  $I_1, \dots, I_n$ , implemented by  $C$ , with

$$\begin{aligned} M_C &= C.ownedMembers \cup \{m \in C.superType.members \mid m.acc > \text{private}\} \\ M_I &= \bigcup_{i=1}^n I_i.members \end{aligned}$$

Note that these sets already contain only non-private data fields.

#### 5.2.8.2.1. Member Consumption

##### Member Consumption and Implementation

##### Definition: Member Consumption and Implementation

A member  $M$  defined in an interface  $I$  is *consumed* by an implementor  $C$ , if it becomes a member of the class, that is,  $M \in C.members$ .

A member  $\text{M}$  is consumed if there is no member defined in the implementor with the same name and if there is no non-private, non-abstract member with that name inherited by the implementor from its superclass.<sup>11</sup>

If the implementor defines the member itself, then the member is implemented rather than consumed.

The concrete rules are described in the following;

It is not always possible to directly consume a member. In general, a rather conservative strategy is used: if two implemented interfaces define the same (non-abstract) member then the implementor must redefine the member in order to solve conflicts. Even if the two conflicting members have the same types, the implementor must redefine them as we generally assume semantic differences which the consumer has to be aware of. Data fields defined in interfaces, in particular, are assumed to be concrete. It is not, therefore, possible to consume a field defined in two implemented interfaces.

### Consumption of Interface Members

#### Requirement: IDE-73: Consumption of Interface Members (ver. 1)

Given a classifier  $\text{C}$ <sup>12</sup>, and interfaces  $I_1, \dots, I_n$  implemented (or extended) by  $\text{C}$ , and sets  $M_C$  and  $M_I$  as defined in [interface\_and\_class\_member\_sets]. A non-static member  $\text{M}$  defined in any interface  $I_i$  is merged into the consumer ( $\text{C}$ ), if for all other (possible) members  $M'$  of  $\text{C}$

$$\forall M' \in M_C \cup M_I \setminus \{M\} : M.name = M'.name \wedge \neg M'.static$$

the following constraints hold:

1. The other member's meta type matches the meta type of the merge candidate:

$$\begin{aligned} \mu(M) = \text{Method} &\rightarrow \mu(M') = \text{Method} \\ \mu(M) \neq \text{Method} &\rightarrow \mu(M') \in \{\text{Field, FieldAccessor}\} \end{aligned}$$

2. The other member is abstract and not owned by the consumer:

$$\begin{aligned} \mu(M) = \mu(M') \vee \mu(M) = \text{Field} \\ \rightarrow M'.abstract \wedge M' \notin C.ownedMembers \end{aligned}$$

3. The merge candidate's access modifier is not less than the modifier of the other member:

$$\begin{aligned} \mu(M) = \mu(M') \vee \mu(M) = \text{Field} \\ \rightarrow M.access \geq M'.access \end{aligned}$$

4. The merge candidate's type compatible with the other member:

$$\begin{aligned} \mu(M) \in \{\text{Method, Getter, Field}\} \wedge \mu(M') \neq \text{Setter} &\rightarrow \Gamma \vdash M < : M' \\ \mu(M) \in \{\text{Setter, Field}\} \wedge \mu(M') \neq \text{Getter} &\rightarrow \Gamma \vdash M' < : M \end{aligned}$$

### 5.2.8.2.2. Member Implementation

#### Implementation of Interface Members

#### Requirement: IDE-74: Implementation of Interface Members (ver. 1)

For any non-static abstract member  $\text{M}$  defined in an interface  $\text{I}$  implemented (or extended) by a classifier  $\text{C}$ ,  $\text{M}$  must be accessible from  $\text{C}$  and one or two member(s) in  $\text{C}$  must exist which are implementation-compatible with  $\text{M}$ . The implementing member(s) must be declared as override if they are directly defined in the consumer.

1.  $\text{M}$  must be accessible from  $\text{C}$ .
2. An implementation-compatible member  $M'$  must exist in  $\text{C}$ :

- a. if  $\text{M}$  is not a field:

$$\begin{aligned} \mu(M) \neq \text{Field} &\rightarrow \\ \exists M' \in C.members: & \\ \text{overrideCompatible}(M', M) & \end{aligned}$$

<sup>11</sup> There had been the idea of preventing static members of being consumed. However, this would break the type subtype relation.  
<sup>12</sup>  $\text{C}$  could either be a class or an interface.

$\wedge (M' \in C.ownedMembers \rightarrow M'.override)$
b. if $M$ is a field, then either an implementation-compatible field $F'$ or accessor pair $G', S'$ must exist:
$\mu(M) = \text{Field} \rightarrow$
$\exists F' \in C.fields:$
$overrideCompatible(F', M)$
$\wedge (F' \in C.ownedMembers \rightarrow F'.override)$
v
$\exists G' \in C.getters, S' \in C.setters:$
$overrideCompatible(G', M)$
$\wedge overrideCompatible(S', M)$
$\wedge (G' \in C.ownedMembers \rightarrow G'.override)$
$\wedge (S' \in C.ownedMembers \rightarrow S'.override)$

Methods defined in interfaces are automatically declared abstract if they do not provide a default implementation. This can also be expressed explicitly via adding the `abstract` modifier. If a class implementing an abstract interface does not implement a method declared in the interface, the class needs to be declared abstract (cf. [Section 5.1.3.4, “Abstract Classes”](#)).

Consequences for method implementation:

1. It may be required for the implementor to explicitly define a method in order to solve type conflicts produced by methods of different interfaces with same name but different signatures.
  2. Methods in an implementor cannot decrease the accessibility of methods from implemented interfaces, that is
- $$\forall M \in C.methods, M' \in I_i.methods (i = 1 \dots n):$$
- $$M.name = M'.name \rightarrow M.acc \neq private \Rightarrow M.acc \geq M'.acc$$
3. Methods in the implementor must be a supertype <sup>13</sup> of methods from implemented interfaces. That is to say the implemented methods are override-compatible.
  4. There may be several methods  $M_1, \dots, M_n$  defined in different implemented interfaces and a single owned method  $M'$  in  $M_C$ . In this case, the above constraints must hold for all methods. In particular,  $M'$ 's signature must conform to all conflicting methods' signatures. This is possible by using union types for the arguments and an intersection type as return type. Such a method  $M'$  is said to *resolve* the conflict between the implemented (and also inherited) methods.
  5. Since abstract methods may become part of the implementor methods, the implementor must either define these methods or it must be declared abstract itself. Since interfaces are abstract by default, responsibility for implementing abstract methods is passed on to any implementor of interfaces.
  6. If two implemented interfaces provide (non-abstract) members with the same name, they are not automatically consumed by the implementor even if the types would be similar. In these cases, the implementor has to redefine the members in order to be aware of possible semantic differences.

There is currently no separate annotation to indicate that methods are implemented or overridden in order to solve conflicts. We always use the `@Override` annotation.

#### Example 5.19. Method Consumption

[Table 5.3, “Consumption of methods”](#) shows simple examples of above rules. Assuming that `class C extends super class S and implements interface I1 and I2`:

```
class C extends S implements I1, I2 { ... }
```

The columns describe different scenarios in which a method (with same name) is defined in different classifiers. We assume that the defined methods are always non-abstract (i.e. have default implementations), non-private and have the same signature. The last row shows which method will be actually used in class `C`. If the method is defined in class `C`, and if this method is printed bold, then this means that the method is required to be defined in `C` in order to solve conflicts.

[Table 5.3. Consumption of methods](#)

Interface I1	$M_{I1}$	$M_{I1}$	$M_{I1}$	$M_{I1}$	$M_{I1}$	$M_{I1}$

<sup>13</sup> As defined in [Section 6.1, “Function Type”](#) for function types.

Interface <code>I2</code>			$M_{I2}$		$M_{I2}$	$M_{I2}$
<code>class S</code>				$M_S$	$M_S$	$M_S$
<code>class C</code>		$M_C$	$M_C$			$M_C$
<code>∈ C.members</code>	$M_{I1}$	$M_C$	$M_C$	$M_S$	$M_S$	$M_C$

**Consuming Field Initializers** Aside from the fields themselves, an implementor *always* consumes the field initialization if the field is consumed – this is how the consumption is noticed at runtime.

#### Example 5.20. Field and Field Initializer Consumption

```
/* XPECT output ~~~
<==
stdout:
s: C , t: D ,u: I1 ,v: I2
stderr:
==>
~~~ */

interface IO {
    v: string = "IO";
}

interface I1 {
    s: string = "I1";
    t: string = "I1";
    u: string = "I1";
}

interface I2 extends I1, IO {
    @Override
    t: string = "I2";
    @Override
    v: string = "I2";
}

class C {
    s: string = "C";
}

class D extends C implements I1, I2 {
    @Override
    t: string = "D";
}

var d = new D();

console.log(
    "s:", d.s, ", t:", d.t, ",u:", d.u, ",v:", d.v
)
```

We expect the following output (for each field):

- `d.s = "C"` : `s` is inherited from `C`, so it is not consumed from `I1` (or `I2`). Consequently, the initializer of `s` in `C` is used.
- `d.t = "D"` : `t` is defined in `D`, solving a conflict stemming from the definition of `t` in `I1` and `I2`. Thus, the initializer of `t` in `D` is used.
- `d.u = "I1"` : `u` is only defined in `I1`, thus the initializer defined in `I1` is used.
- `d.v = "I2"` : `v` is overridden in `I2`, so is the field initializer. This is why `d.v` must be assigned to `I2` and not `IO`.

## 5.3. Structural Typing

In general, N4JS uses nominal typing. This is to say that a duck is a duck only if it is declared to be a duck. In particular when working with external APIs, it is more convenient to use structural or duck typing. That is, a thing that can swim and quacks like a duck, is a duck.

Interfaces or classes can be used for this purpose with a *typing strategy modifier*. Given a type  $T$ , the simple `~` (tilde) can be added to its declaration (on definition-site) or in a reference (on use-site) to indicate that the type system should use structural typing rather than nominal typing.<sup>14</sup> This means that some other type must provide the same members as type  $T$  to be deemed a structural

<sup>14</sup> This kind of typing is used by TypeScript only. By defining a structural typed classifier or reference, it basically behaves as it would behave – without that modifier – in TypeScript.

subtype. However, the operator cannot be used anymore with the type or reference as this operator relies on the declaration information (or at least the closest thing available at runtime). In this case,  $T$  is, therefore, always a structural subtype of  $\sim T$ .

Sometimes it is convenient to refer only to the fields of a classifier, for example when the initial field values are to be provided in a variable passed to the constructor. In that case, the type can be modified with  $\sim\sim$  (two tildes). This is only possible on use-site, i.e. on type references. Furthermore, only on the use-site, it is possible to consider only either readable or writable or fields by using the read-only  $\sim r \sim$  or write-only  $\sim w \sim$  structural field typing. For initialization blocks, it is even possible to use structural initializer field typing via the  $\sim i \sim$  operator.

### 5.3.1. Syntax

Structural typing is specified using the typing strategy modifier. There are two modifiers defined; one for definition-site and one for use-site structural typing.

#### Structural Type Operator and References

```

TypingStrategyUseSiteOperator returns TypingStrategy:
  '~~' ('~~' | STRUCTMODSUFFIX)?;

TypingStrategyDefSiteOperator returns TypingStrategy:
  '~~';

terminal STRUCTMODSUFFIX:
  ('r' | 'i' | 'w') '~~'
;

ParameterizedTypeRefStructural returns ParameterizedTypeRefStructural:
  definedTypingStrategy=TypingStrategyUseSiteOperator
  declaredType=[Type/TypeReferenceName]
  (=> '<' typeArgs+=TypeArgument (',' typeArgs+=TypeArgument)* '>')?
  (=> 'with' '{' astStructuralMembers+=TStructMember* '}')?
;

ThisTypeRefStructural returns ThisTypeRefStructural:
  definedTypingStrategy=TypingStrategyUseSiteOperator
  'this'
  ('with' '{' astStructuralMembers+=TStructMember* '}')?
;

```

### 5.3.2. Definition Site Structural Typing

An interface or class can be defined to be used with structural typing by adding the structural modifier to its definition (or, in case of external classes, to the declaration). This changes the default type system strategy from nominal to structural typing for that type. That means that all types with the same members as the specified type are subtypes of that type, except for subtypes of `N4Object`. In the latter case, programmers are enforced to nominal declare the type relation.

If a type  $T$  is declared as structural at its definition,  $T.defStructural$  is true.

#### Definition Site Structural Typing

##### Requirement: IDE-75: Definition Site Structural Typing (ver. 1)

1. The structurally defined type cannot be used on the right hand side of the `instanceof` operator:

$$x \text{ instanceof } T \Rightarrow \neg T.defStructural$$

2. A type  $X$  is a subtype of a structurally defined type  $T$  either

- a. if it is not a subtype of `N4Object`<sup>15</sup> but it contains all public, non-static members of that type

$$\begin{aligned}
 & X \not\prec \{N4Object\} \quad T.defStructural \\
 & \forall m \in T.members, m.acc = \text{public}, \neg m.static, m \neq T.ctor: \\
 & \exists m' \in X.members: \\
 & \quad m'.acc = \text{public} \wedge \neg m'.static \wedge m'.name = m.name \\
 & \quad \wedge \Gamma \vdash m' < : m \\
 & \quad \frac{\wedge \mu(m) = \text{Field} = \Gamma \vdash m < : m'}{\Gamma \vdash X < : T}
 \end{aligned}$$

or

<sup>15</sup>We enforce programmers of N4JS to use nominal typing, therefore, it is not possible to bypass that principle by declaring a type as structural for normally defined classes (except those explicitly derived from `N4Object`).

- b. if it is a subtype of `N4Object` which explicitly extends or implements the structurally defined type.

$$\frac{X < : \{N4Object\} \quad T.\text{defStructural} \quad T \in X.\text{superTypes}^*}{\Gamma \vdash X < : T}$$

- c. A structurally defined type  $T$  is implicitly derived from `Object` if no other type is specified. In particular, a structurally defined type must not be inherited from

$$\frac{T.\text{defStructural}}{\Gamma \vdash T < : Object}$$

$$T.\text{defStructural} = \Gamma \vdash T < : N4Object \wedge N4Object \notin T.\text{superTypes}^*$$

#### Example 5.21. Declaration Site Structural Typing

The following snippet demonstrates the effect of definition-site structural types by comparing them to nominal declared types:

##### Declaration Site Structural Typing

```
interface ~Tilde { x; y; }
interface Nominal { x; y; }
class C { public x; public y; }
class D extends C implements Tilde { }

function f(p: Tilde) {}
function g(p: Nominal) {}

f(new C());           // error: nominal typing, C does not implement ~Tilde
f(new D());           // ok, D is a nominal subtype (as it implements Tilde)
f({x:10,y:10});      // ok: Tilde is used with structural typing for non-N4-classifiers
```

Definition site structural typing may lead to unexpected results. For example;

```
class C{}
class ~E extends C{}
```

It may be unexpected, but `E` is not a subtype of `C`, i.e.  $E \not< : C$ . E.g., `instanceof` won't work with `E`, while it works with `C`!

### 5.3.3. Use-Site Structural Typing

Use-site structural typing offers several typing strategy modifiers to define the accessibility of public properties of classes and interfaces. They can be used e.g. on variable declarations like this: `var c : ~C`. The table [Table 5.4, “Available Fields of Structural Types”](#) shows which properties of structural types can be accessed in the different type strategies. For example, when using the write-only structural strategy (i.e.  $\sim w \sim X$ ), only the writeable fields, can be accessed for writing. In the table, the term field to both, public datafields and accessors of type  $X$ . Non-public properties are never accessible in use-site structural types. In any field-structural type, methods of the referenced nominal type  $X$  are not available. The initializer structural typing provides readable fields for every writeable field of the references type.

**Table 5.4. Available Fields of Structural Types**

Property of $X$	$\sim X$	$\sim\sim X$	$\sim r \sim X$	$\sim w \sim X$	$\sim i \sim X$
public method		$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
public writable field			$\emptyset$		$\emptyset$
public readable field				$\emptyset$	writable fields

Multiple structural typing strategies can be nested when there are multiple use sites, like in the example [Example 5.22, “Nested Structural Typing Strategies”](#) below at the locations ST1 and ST2. In the example, the datafield `a.field` has the nested structural type  $\sim r \sim \sim i \sim A$  and thus the datafield `a.field.df` is readable. Nested structural types are evaluated on the fly when doing subtype checks.

#### Example 5.22. Nested Structural Typing Strategies

```
class A {
    public df : string;
}
interface I<T> {
    public field : ~r~T; // ST1
```

```

}
var a : ~i~A; // ST2

```

The following example demonstrates the effect of the structural type modifiers:

**Example 5.23. Effect of structural type modifiers on use-site**

Let's assume the type defined on the left. The following *pseudo* code snippets explicitly list the type with its members virtually created by a structural modifier. Note that this is pseudo code, as there are no real or virtual types created. Instead, only the subtype relation is defined accordingly:

Effect of structural type modifiers on use-site

Effect of structural type modifiers on use-site		
Type	Structural Type	Structural Field Type
<pre> var c:C  class C {     private x;     public y;     public f()     private g()     public get z():Z     public set z(z:Z) }  interface I {     a;     func(); } </pre>	<pre> var cstructural:~C  class cstructural {     public y;     public f()      public get z():Z     public set z(z:Z) }  interface ~I {     a;     func(); } </pre>	<pre> var cfields:~~C  class cfields {      public y;      public get z():Z     public set z(z:Z) }  interface ~~I {     a; } </pre>
Structural Read-only Field Type	Structural Write-only Field Type	Structural Initializer Field Type

Note that even if a type is defined without the structural modifier, it is not possible to use `instanceof` for variables declared as structural, as shown in the next example:

Type	Structural Type	Structural Field Type
<pre>class C {...} interface I {...}  foo(c: C, i: I) {     c instanceof C; // ok     c instanceof I; // ok }</pre>	<pre>class C {...} interface I {...}  foo(c: ~C, i: ~I) {     c instanceof C; // error     c instanceof I; // error }</pre>	<pre>class C {...} interface I {...}  foo(c: ~~C, i: ~~I) {     c instanceof C; // error     C instanceof I; // error }</pre>

- If a type is referenced with the structural type modifier `~`, the property `T.refStructural` is true.
- If a type is referenced with the structural field type modifier `~~`, the property `T.refStructuralField` is true.
- If a type is referenced with the structural read-only field type modifier `~r~`, the property `T.refStructuralReadOnlyField` is true.
- If a type is referenced with the structural write-only field type modifier `~w~`, then the property `T.refStructuralWriteOnlyField` is true.
- If a type is referenced with the structural initializer field type modifier `~i~`, then the property `T.refStructuralInitField` is true.

We call the following:

- $T$  the (nominal) type  $T$ ,
- $\sim T$  the structural version of  $T$ ,
- $\sim\sim T$  the structural field version of  $T$ ,
- $\sim r\sim T$  the structural read-only field,
- $\sim w\sim T$  the structural write-only field and
- $\sim i\sim T$  the structural initializer field version of  $T$ .

## Use-Site Structural Typing

### Requirement: IDE-76: Use-Site Structural Typing (ver. 1)

1. The structural version of a type is a supertype of the nominal type:

$T < : \sim T$  2. The structural field version of a type is a supertype of the structural type:

$\sim T < : \sim\sim T$  3. The structural read-only field version of a type is a supertype of the structural field type:

$\sim\sim T < : \sim r\sim T$  4. The structural write-only field version of a type is a supertype of the structural field type:

$\sim\sim T < : \sim w\sim T$  5. The structural (field) version of a type cannot be used on the right hand side of the `instanceof` operator:

+

$$\begin{aligned} x \text{ instanceof } E \Rightarrow & \Gamma \vdash E : T \\ \Rightarrow & \neg(T.\text{refStructural} \\ & \vee T.\text{refStructuralField} \\ & \vee T.\text{refStructuralReadOnlyField} \\ & \vee T.\text{refStructuralWriteOnlyField} \\ & \vee T.\text{refStructuralInitField}) \end{aligned}$$

+ That is, the following code will always issue an error: `x instanceof ~T` <sup>16</sup>. A type  $X$  is a subtype of a structural version of a type  $\sim T$ , if it contains all public, non-static members of the type  $T$ : <sup>17</sup>

+

$\forall m \in T.\text{members}, m.\text{owner} \notin \text{N40bject}, m.\text{acc} = \text{public}, \neg m.\text{static}, m \neq T.\text{ctor}$ :

$$\begin{aligned} \exists m' \in X.\text{members}: \\ m'.\text{acc} = \text{public} \wedge \neg m'.\text{static} \wedge m'.\text{name} = m.\text{name} \end{aligned}$$

$$\frac{\wedge \Gamma \vdash m' < : \Gamma \vdash m}{\Gamma \vdash X < : \Gamma \vdash \sim T}$$

<sup>16</sup> Since this is already prevented by the parser (the tilde is interpreted as an unary operator), error messages are likely to look a little strange.

<sup>17</sup> Note that due to this relaxed definition (compared with definition-site structural types) it is possible to pass an `N40bject` instance to a function of method with a declared structural type parameter.

7. A type  $X$  is a subtype of a structural field version of a type  $\sim\sim T$ , if it contains all public, non-static fields of the type  $T$ . Special cases regarding optional fields are described in [Optional Fields](#).

+

$$\frac{\begin{array}{c} \forall m \in T.fields, m.owner \notin \text{N4Object}, m.acc = \text{public}, \neg m.static \\ \vee \exists m' \in X.fields: \\ \quad m'.acc = \text{public} \wedge \neg m'.static \wedge m'.name = m.name \\ \wedge \Gamma \vdash m:T_m \wedge \Gamma \vdash m:T_{m'} \wedge T_m = T_{m'} \\ \wedge m'.assignability \geq m.assignability \end{array}}{\Gamma \vdash X < : \sim\sim T}$$

+ 8. A type  $X$  is a subtype of a structural read-only field version of a type  $\sim r\sim T$ , if it contains all public and non-static readable fields of the type  $T$ . Special cases regarding optional fields are described in [Optional Fields](#).

+

$$\frac{\begin{array}{c} \forall m \in T.fields \cup T.getters, m.owner \notin \text{N4Object}, m.acc = \text{public}, \neg m.static \\ \vee \exists m' \in X.fields \cup X.getters: \\ \quad m'.acc = \text{public} \wedge \neg m'.static \wedge m'.name = m.name \\ \wedge \Gamma \vdash m:T_m \wedge \Gamma \vdash m:T_{m'} \wedge T_m = T_{m'} \\ \wedge m'.assignability \geq m.assignability \end{array}}{\Gamma \vdash X < : \sim r\sim T}$$

+ 9. A type  $X$  is a subtype of a structural write-only field version of a type  $\sim w\sim T$ , if it contains all public and non-static writable fields of the type  $T$ . Special cases regarding optional fields are described in [Optional Fields](#).

+

$$\frac{\begin{array}{c} \forall m \in T.fields \cup T.setters, m.owner \notin \text{N4Object}, m.acc = \text{public}, \neg m.static, \neg m.final \\ \vee \exists m' \in X.fields \cup X.setters: \\ \quad m'.acc = \text{public} \wedge \neg m'.static \wedge m'.name = m.name \\ \wedge \Gamma \vdash m:T_m \wedge \Gamma \vdash m:T_{m'} \wedge T_m = T_{m'} \\ \wedge m'.assignability \geq m.assignability \end{array}}{\Gamma \vdash X < : \sim w\sim T}$$

+ 10. A type  $X$  is a subtype of a structural field version of a type  $\sim\sim this$ , if it contains all public, non-static fields, either defined via data fields or field get accessors, of the inferred type of `this`. Special cases regarding optional fields are described in [Optional Fields](#).

+

$$\frac{\begin{array}{c} \Gamma \vdash this:T \\ \forall m \in T.fields \cup T.setters, m.owner \notin \text{N4Object}, m.acc = \text{public}, \neg m.static \\ \vee m.expr \neq \text{null} \\ \vee \exists m' \in X.fields \cup X.getters: \\ \quad m'.acc = \text{public} \wedge \neg m'.static \wedge m'.name = m.name \\ \wedge \Gamma \vdash m' < : m \wedge m'.assignability \geq m.assignability \end{array}}{\Gamma \vdash X < : \sim\sim this}$$

+ 11. A structural field type  $\sim\sim T$  is a subtype of a structural type  $\sim X$ , if  $\sim X$  only contains fields (except methods inherited from `Object`) and if  $\sim\sim T < : \sim\sim X$ .

+

$$\frac{X.methods \setminus \text{Object}.methods = \emptyset \wedge \Gamma \vdash \sim\sim T < : \sim\sim X}{\Gamma \vdash \sim\sim T < : \sim X}$$

+ 12. Use-site structural typing cannot be used for declaring supertypes of classes or interfaces. That is to say that structural types cannot be used after `extends`, `implements` or `with` in type declarations <sup>18</sup>.

Note that all members of `N4Object` are excluded. This implies that extended reflective features (cf. [Section 7.4, “Reflection meta-information”](#)) are not available in the context of structural typing. The `instanceof` operator is still working as described in [Section 8.1.16, “Relational Expression”](#), in that it can be used to check the type of an instance.

<sup>18</sup>This is already constrained by the grammar.

If a type  $X$  is a (nominal) subtype of  $T$ , it is, of course, also a subtype of  $\sim T$ :

$$\frac{\Gamma \vdash X \leq : \Gamma \vdash T}{\Gamma \vdash X \leq : \Gamma \vdash \sim T}$$

This is only a shortcut for the type inference defined above.

### Definition and Use-Site Precedence

#### Requirement: IDE-77: Definition and Use-Site Precedence (ver. 1)

If a type is structurally typed on both definition and use-site, the rules for use-site structural typing ([Req-IDE-76]) are applied.

#### Example 5.24. Use-Site Structural Typing

The following example demonstrates the effect of the structural (field) modifier, used in this case for function parameters.

```
interface I { public x: number; public foo(); }
class C { public x: number; public foo() {} }

function n(p: I) {}
function f(p: ~I) {}
function g(p: ~~I) {}

n(new C());      // error: nominal typing, C does not implement I
f(new C());      // ok: C is a (structural) subtype of ~I
f({x:10});        // error, the object literal does not provide function foo()
g({x:10});        // ok: object literal provides all fields of I
```

#### Example 5.25. Structural types variable and instanceof operator

It is possible to use a variable typed with a structural version of a type on the left hand side of the `instanceof` operator, as demonstrated in this example:

```
class C {
    public x;
    public betterX() { return this.x | 1; }
}

function f(p: ~~C) {
    if (p instanceof C) {
        console.log((p as C).betterX);
    } else {
        console.log(p.x | 1);
    }
}
```

The following table describes the member availability of `x` in various typing scenarios. Such as `~~x`, `~r~x`, `~w~x`, `~i~x`.

Table 5.5. Member Availability in various Typing Scenarios

Member of type <code>X</code>	<code>~~x</code>	<code>~r~x</code>	<code>~w~x</code>	<code>~i~x</code>
<code>private m0;</code>	—	—	—	—
<code>public set m1(m) { }</code>	write	—	write	read
<code>public get m2() {...}</code>	read	read	—	
<code>public m3;</code>	read/write	read	write	read
<code>public m4 = 'init.m4';</code>	read/write	read	write	read ?
<code>public m5: any?;</code>	read?/write	read?	write	read?
<code>@Final public m6 = 'init.m6';</code>	read	read		
<code>@Final public m7;</code>	read	read		read
<code>public get m8() {...}</code>	read/write	read	write	read
<code>public set m8(m) { }</code>				

### 5.3.4. Structural Read-only, Write-only and Initializer Field Typing

Structural read-only, write-only and initializer field typings are extensions of structural field typing. Everything that is defined for the field structural typing must comply with these extension field typings. For the read-only type, readable fields (mutable and `@Final` ones) and setters are considered, for the write-only type; only the setters and mutable fields are considered. Due to the hybrid nature of the initializer type it can act two different ways. To be more precise, a type  $X$  (structural initializer field type) is a supertype of  $Y$  (structural initializer field type) if for each public, non-static, non-optional writable field (mutable data field of setter) of  $X$ , there is a corresponding, public, non-static readable field of  $Y$ . All public member fields with `@Final` annotation are considered to be mandatory in the initializer field typing `@Spec` constructors. The already-initialized `@Final` fields can be either omitted from, or can be re-initialized via, an initializer field typing `@Spec` style constructor.

**Example 5.26. Subtype relationship between structural field typing**

```
class A1 {
    public s: string;
}

class A2 {
    public set s(s: string) { }
    public get s(): string { return null; }
}

class A3 {
    @Final public s: string = null;
}

class A4 {
    public get s(): string { return null; }
}

class A5 {
    public set s(s: string) { }
}
```

	A1	~A1	~~A1	~r~A1	~r~A2	~r~A3	~r~A4	~r~A5	w~A1	w~A2	w~A3	w~A4	w~A5	i~A1	i~A2	i~A3	r~A4	r~A5
A1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
~A1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
~~A1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	
~r~A1				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A2				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A3				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A4				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A5									✓		✓	✓			✓	✓		
~w~A1									✓	✓	✓	✓	✓			✓	✓	
~w~A2									✓	✓	✓	✓	✓			✓	✓	
~w~A3									✓			✓	✓			✓	✓	
~w~A4									✓			✓	✓			✓	✓	
~w~A5									✓	✓	✓	✓	✓			✓	✓	
~i~A1				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~i~A2				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~i~A3				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A4				✓	✓	✓	✓	✓			✓	✓		✓	✓	✓	✓	✓
~r~A5									✓		✓	✓				✓	✓	

### 5.3.5. Public Setter Annotated With `ProvidesInitializer`

Public setters with `ProvidesInitializer` annotation can declare optional fields implemented by means of field accessors instead of data fields. Data fields with an initializer are treated as optional in the initializer field type.

It is important to note that it is valid to use the `ProvidesInitializer` annotation for setters in `n4js` files and not only definition files.

#### Example 5.27. Setters with `@ProvidesInitializer` treated as optional

```
class C {
    private _y: int = 1;

    public get y() { return this._y; }
    @ProvidesInitializer
    public set y(v: int) { this._y = v; }

    public constructor(@Spec spec: ~i~this) { }
}

console.log(new C().y); // 1
console.log(new C({y: 42}).y); // 42
```

### 5.3.6. Structural Types With Optional Fields

Public optional fields become a member of the structural (field) type as well. To ensure the overall type safety, the semantics of optionality (e.g. on or off) depends on the context, in which the optional fields are currently being used (See [Optional Fields](#)).

### 5.3.7. Structural Types With Access Modifier

The access modifier of the subtype have to provide equal or higher visibility.

#### Example 5.28. Access modifier in structural typing

```
class C {
    public s: number;
}

class D {
    project s: number;
```

```

    }
    function f(c: ~C) {}
f(new D()); // error: D is no (structural) subtype of ~C, as visibility of s in D is lower
    function g(d: ~D) {}
g(new C()); // ok: C is a (structural) subtype of ~D, as visibility of s in C is greater-than-or-equal to s in D
  
```

### 5.3.8. Structural Types With Additional Members

It is possible to add additional members when structurally referencing a declared type.

#### 5.3.8.1. Syntax

```

TStructMember:
  TStructGetter | TStructGetterES4 | TStructSetter | TStructMethod | TStructMethodES4 | TStructField;

TStructMethod:
  =>
  ({TStructMethod} ('<' typeVars+=TypeVariable (',' typeVars+=TypeVariable)* '>')?
  returnTypeRef=TypeRef name=TypesIdentifier '('
  (fpars+=TAnonymousFormalParameter (',' fpars+=TAnonymousFormalParameter)*)? ')'
  ';'?;

TStructMethodES4 returns TStructMethod:
  =>
  ({TStructMethod} ('<' typeVars+=TypeVariable (',' typeVars+=TypeVariable)* '>')?
  name=TypesIdentifier '('
  (fpars+=TAnonymousFormalParameter (',' fpars+=TAnonymousFormalParameter)*)? ')'
  (':' returnTypeRef=TypeRef)?
  ';'?;

TStructField:
  (
    typeRef=TypeRef name=TypesIdentifier
    | name=TypesIdentifier (':' typeRef=TypeRef)?
  )
  ';'?;

TStructGetter:
  => ({TStructGetter}
  declaredTypeRef=TypeRef
  'get'
  name=TypesIdentifier)
  '(' ')' ';'?;

TStructGetterES4 returns TStructGetter:
  => ({TStructGetter}
  'get'
  name=TypesIdentifier)
  '(' ')' (':' declaredTypeRef=TypeRef)? ';'?;

TStructSetter:
  => ({TStructSetter}
  'set'
  name=TypesIdentifier)
  (':' fpar=TAnonymousFormalParameter ')') ';'?;

TAnonymousFormalParameter:
  typeRef=TypeRef variadic?='...'? name=TIdentifier?
  | variadic?='...'? (=> name=TIdentifier ':') typeRef=TypeRef;
  
```

##### 5.3.8.1.1. Semantics

Additional structural members

**Requirement: IDE-78:** Additional structural members (ver. 1)

It is only possible to add additional members to a type if use-site structural typing is used.

The following constraints must hold:

1. For all additional members defined in a structural type reference, the constraints for member overriding ([Req-IDE-72]) apply as well.
2. All additional members have the access modifier set to `public`.
3. Type variables must not be augmented with additional structural members.

Additional fields may be declared optional in the same way as fields in classes. The rules given in [Section 5.3.6, “Structural Types With Optional Fields”](#) apply accordingly. Consider the following example:

**Example 5.29. Additional optional members in structural typing**

```
class C {
    public f1: number;
}

var c1: ~C with { f3: string; } c1;
var c2: ~C with { f3: string?; } c2;

c1 = { f1:42 }; // error: "~Object with { number f1; } is not a subtype of ~C with { string f3; }."
c2 = { f1:42 }; // ok!!
```

Augmenting a type variable T with additional structural members can cause collisions with another member of a type argument for T. Hence, type variables must not be augmented with additional structural members like in the following example.

**Example 5.30. Forbidden additional structural members on type variables**

```
interface I<T> {
    public field : ~T with {prop : int} // error "No additional structural members allowed on type variables."
}
```

Using an additional structural member on a type variable T could be seen as a constraint to T. However, constraints like this should be rather stated using an explicit interface like in the example below.

**Example 5.31. Use explicitly defined Interfaces**

```
interface ~J {
    prop : int;
}
interface II<T extends J> {
```

# Chapter 6. Functions

Functions, be they function declarations, expressions or even methods, are internally modeled by means of a function type. In this chapter, the general function type is described along with its semantics and type constraints. Function definitions and expressions are then introduced in terms of statements and expressions. Method definitions and special usages are described in [Section 5.2.3, “Methods”](#).

## 6.1. Function Type

A function type is modeled as `Object` (see [[ECMA11a\(p.S13, p.p.98\)](#)]) in ECMAScript.

Function types can be defined by means of;

- A function object ([Section 4.7.3, “Function-Object-Type”](#)).
- A function type expression ([Section 4.2, “Type Expressions”](#)).
- A function declaration ([Section 6.2.1, “Function Declaration”](#)).
- A method declaration ([Section 5.2.3, “Methods”](#)).

### 6.1.1. Properties

In any case, a function type declares the signature of a function and allows validation of calls to that function. A function type has the following properties:

<code>typePars</code>	(0-indexed) list of type parameters (i.e. type variables) for generic functions.
<code>fpars</code>	(0-indexed) list of formal parameters.
<code>returnType</code>	(possibly inferred) return type (expression) of the function or method.
<code>name</code>	Name of function or method, may be empty or automatically generated (for messages).
<code>body</code>	The body of the function, it contains statements <i>stmts</i> . The body is null if a function type is defined in a type expression, and it is the last argument in case of a function object constructor, or the content of the function definition body.

Additionally, the following pseudo properties for functions are defined:

<code>thisTypeRef</code>	The this type ref is the type to which the <code>this</code> -keyword would be evaluated if used inside the function or member. The inference rules are described in <a href="#">Section 2.1.2, “This Keyword”</a> .
<code>fpars</code>	List of formal parameters and the this type ref. This is only used for sub typing rules. If <code>this</code> is not used inside the function, then <code>any</code> is set instead of the inferred thisTypeRef to allow for more usages. The property is computed as follows:  $tfpars = \text{if... } \text{this is used or explicitly declared } \text{then... } thisTypeRef + fpars \text{ else... } any + fpars$

Parameters (in *pars*) have the following properties:

<code>name</code>	Name of the parameter.
<code>type</code>	Type (expression) of the parameter. Note that only parameter types can be variadic or optional.

The function definition can be annotated similar to [Section 5.2.3, “Methods”](#) except that the `final` and `abstract` modifiers aren't supported for function declarations. A function declaration is always final and never abstract. Also, a function has no property advice set.

## Semantics

### Function Type

#### Requirement: IDE-79: Function Type (ver. 1)

Type Given a function type *F*, the following constraints must be true:

1. Optional parameters must be defined at the end of the (formal) parameter list. In particular, an optional parameter must not be followed by a non-optional parameter:

$$F.fpars_i; optional \rightarrow \#k > i : \neg F.fpars_k.optvar$$

2. Only the last parameter of a method may be defined as variadic parameter:

$$F.fpars_i; variadic \rightarrow i = |F.fpars| - 1$$

3. If a function explicitly defines a return type, the last statement of the transitive closure of statements of the body must be a return statement:

$$\begin{aligned} F.typeRef \neq \text{Undefined} \rightarrow \\ |f.bodystmts| > 0 \\ \wedge f.bodystmts^*_{|f.bodystmts|-1} \text{ is a ReturnStatement} \end{aligned}$$

4. If a function explicitly defines a return type, all return statements must return a type conform to that type:

$$\begin{aligned} F.typeRef \neq \text{Undefined} \\ \Leftrightarrow \\ \forall r \in F.bodystmts, r \text{ is a ReturnStatement:} \\ r.expr \neq \text{null} \wedge [r.expr.typeRef] < \llbracket F.typeRef \rrbracket \end{aligned}$$

## 6.1.2. Type Inference

### Function Type Conformance Non-Parameterized

**Definition: Function Type Conformance Non-Parameterized**

For the given non-parameterized function types  $F_{left}$  with  $F_{left}.tfpars = L_0, L_1, \dots, L_k$  and  $|F_{left}.typesPars| = 0$

$F_{right}$  with  $F_{right}.tfpars = R_0, R_1, \dots, R_n$  and  $|F_{right}.typesPars| = 0$ ,

we say  $F_{left}$  conforms to  $F_{right}$ , written as  $F_{left} < :F_{right}$ , if and only if:

- $F_{right}.returnType = \text{void}$   
 $\vee (F_{left}.returnType = \text{void} \wedge F_{right}.opt)$   
 $\vee (F_{left}.returnType < :F_{right}.returnType \wedge \neg (F_{left}.opt \wedge \neg F_{right}.opt))$
- if  $k \leq n$ :
  - $\forall i, 1 \leq i \leq k: (R_i.opt \rightarrow (L_i.opt \vee L_i.var)) \wedge (R_i.var \rightarrow L_i.var)$
  - $\forall i, 1 \leq i \leq k: L_i > R_i$
  - $L_k.var = \text{true} \rightarrow \forall i, k < i \leq n: L_k > R_i$
  - else ( $k > n$ ):
    - $\forall i, 1 \leq i \leq n: (R_i.opt \rightarrow (L_i.opt \vee L_i.var)) \wedge (R_i.var \rightarrow L_i.var)$
    - $\forall i, 1 \leq i \leq n: L_i > R_i$
    - $\forall n < i \leq k: L_i.opt \vee L_i.var$
    - $R_n.var = \text{true} \rightarrow \forall i, n < i \leq k: L_i > R_n$

Figure 6.1, “Function Variance Chart” shows a simple example with the function type conformance relations.

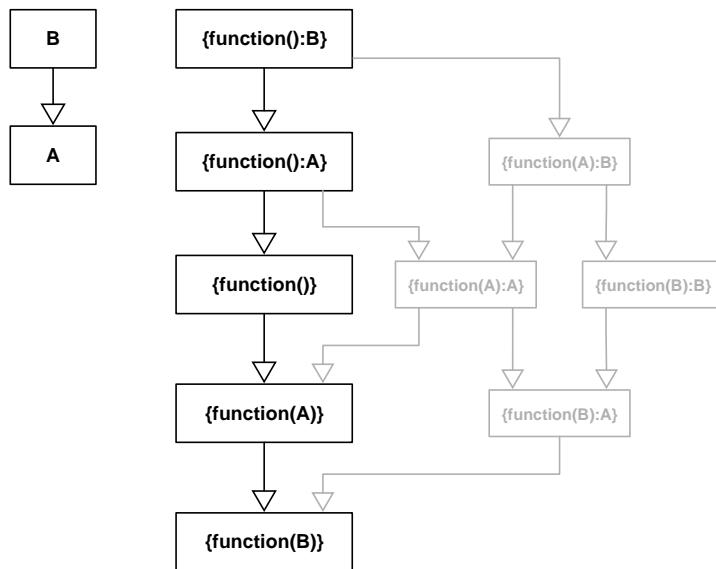


Figure 6.1. Function Variance Chart

`{function()} <: {function(A)}` <: `{function(A, A)}` might be surprising for Java programmers. However, in JavaScript it is possible to call a function with any number of arguments independently from how many formal parameters the function defines.

If a function does not define a return type, `any` is assumed if at least one of the (indirectly) contained return statements contains an expression. Otherwise `void` is assumed. This is also true if there is an error due to other constraint violations.

$$\frac{\text{binds}(f, F) \quad F.\text{returnType}=\text{null} \quad \exists r \in \text{returns}(F) : r.\text{expression} \neq \text{null}}{\Gamma \vdash f'('arglist') : \text{any}}$$

$$\frac{\text{binds}(f, F) \quad F.\text{returnType}=\text{null} \quad \forall r \in \text{returns}(F) : r.\text{expression} \neq \text{null}}{\Gamma \vdash f'('arglist') : \text{void}}$$

with

$$\frac{\{r \in F.\text{body.statements} \mid \mu(r)=\text{ReturnStatement}\} \cup_{s \in F.\text{body.statements}} \text{returns}(s)}{\text{returns}(F):\text{RETS}}$$

$$\frac{\{s \in S.\text{statements} \mid \mu(s)=\text{ReturnStatement}\} \cup_{sub \in S.\text{statements}} \text{returns}(sub)}{\text{returns}(S):\text{RETS}}$$

#### Example 6.1. Function type conformance

The following incomplete snippet demonstrates the usage of two function variables `f1` and `f2`, in which  $[f2] < [f1]$  must hold true according to the aforementioned constraints. A function `bar` declares a parameter `f1`, which is actually a function itself. `f2` is a variable, to which a function expression is assigned. Function `bar` is then called with `f2` as an argument. Thus, the type of `f2` must be a subtype of the `f1`'s type.

```
function bar(f1: {function(A,B):C}) { ... }

var f2: {function(A,B):C} = function(p1,p2){...};
bar(f1);
```

The type of `this` can be explicitly set via the `@This` annotation.

#### Example 6.2. Function Subtyping

```
function f(): A { ... }
function p(): void { ... }

fAny(log: {function():any}) { ... }
fVoid(f: {function():void}) { ... }
fA(g: {function():A}) { ... }

fAny(f); // --> ok      A <: any
fVoid(f); // -->error    A !<: void
fA(f);   // --> ok (easy)  A <: A

fAny(p); // --> ok      void <: any
fVoid(p); // --> ok      void <: void
fA(p);   // --> error    void !<: A
```

#### Example 6.3. Subtyping with function types

If classes A, B, and C are defined as previously mentioned, i.e.  $C < : B < : A$ , then the following subtyping relations with function types are to be evaluated as follows:

<code>{function(B):B} &lt;: {function(B):B}</code>	$\rightarrow \text{true}$
<code>{function():A} &lt;: {function():B}</code>	$\rightarrow \text{false}$
<code>{function():C} &lt;: {function():B}</code>	$\rightarrow \text{true}$
<code>{function(A)} &lt;: {function(B)}</code>	$\rightarrow \text{true}$
<code>{function(C)} &lt;: {function(B)}</code>	$\rightarrow \text{false}$
<code>{function():void} &lt;: {function():void}</code>	$\rightarrow \text{true}$
<code>{function():undefined} &lt;: {function():void}</code>	$\rightarrow \text{true}$
<code>{function():void} &lt;: {function():undefined}</code>	$\rightarrow \text{true } (!)$
<code>{function():B} &lt;: {function():void}</code>	$\rightarrow \text{true } (!)$
<code>{function():B} &lt;: {function():undefined}</code>	$\rightarrow \text{false } (!)$
<code>{function():void} &lt;: {function():B}</code>	$\rightarrow \text{false}$
<code>{function():undefined} &lt;: {function():B}</code>	$\rightarrow \text{true}$

The following examples demonstrate the effect of optional and variadic parameters:

```

{function(A)} <: {function(B)}          -> true
{function(A...)} <: {function(A)}        -> true
{function(A, A)} <: {function(A)}        -> false
{function(A)} <: {function(A,A)}         -> true (!)
{function(A, A...)} <: {function(A)}      -> true
{function(A)} <: {function(A,A...)}       -> true (!)
{function(A, A...)} <: {function(B)}      -> true
{function(A?)} <: {function(A?)}         -> true
{function(A...)} <: {function(A...)}       -> true
{function(A?)} <: {function(A)}          -> true
{function(A)} <: {function(A?)}          -> false
{function(A...)} <: {function(A?)}         -> true
{function(A?)} <: {function(A...)}         -> true (!)
{function(A,A...)} <: {function(A...)}     -> false
{function(A,A?)} <: {function(A...)}       -> true
{function(A...)} <: {function(A?A...)}     -> true
{function(A?A?)} <: {function(A...)}       -> true (!)
{function(A?A?,A?)} <: {function(A...)}    -> true (!)
{function(A?)} <: {function()}            -> true (!)
{function(A...)} <: {function()}           -> true (!)

```

The following examples demonstrate the effect of optional return types:

```

{function():void} <: {function():void}      -> true
{function():X}   <: {function():void}      -> true
{function():X?}  <: {function():void}      -> true
{function():void} <: {function():Y}          -> false
{function():X}   <: {function():Y}          -> X <: Y
{function():X?}  <: {function():Y}          -> false (!)
{function():void} <: {function():Y?}         -> true (!)
{function():X}   <: {function():Y?}         -> X <: Y
{function():X?}  <: {function():Y?}         -> X <: Y
{function():B?}  <: {function():undefined}  -> false (!)
{function():undefined} <: {function():B?}    -> true

```

The following examples show the effect of the `@This` annotation:

```

{@This(A) function():void} <: {@This(X) function():void} -> false
{@This(B) function():void} <: {@This(A) function():void} -> false
{@This(A) function():void} <: {@This(B) function():void} -> true
{@This(any) function():void} <: {@This(X) function():void} -> true
{function():void} <: {@This(X) function():void}           -> true
{@This(A) function():void} <: {@This(any) function():void} -> false
{@This(A) function():void} <: {function():void}            -> false

```

## Function Type Conformance

### Definition: Function Type Conformance

For the given function types

$F_{left}$  with  $F_{left.typePars} = L_0, L_1, \dots, L_k$

$F_{right}$  with  $F_{right.typePars} = R_0, R_1, \dots, R_n$ ,

we say  $F_{left}$  conforms to  $F_{right}$ , written as  $F_{left} <: F_{right}$ , if and only if:

- if  $|F_{left.typePars}| = |F_{right.typePars}| = 0$ :
  - $F_{left} <: F_{right}$  (cf. [Function Type Conformance Non-Parameterized](#))
- else if  $|F_{left.typePars}| > 0 \wedge |F_{right.typePars}| = 0$ :
  - $\exists \theta: (\Gamma \leftarrow \theta) \vdash F_{left} <: F_{right}$  (cf. [Function Type Conformance Non-Parameterized](#))  
(i.e. there exists a substitution  $\theta$  of type variables in  $F_{left}$  so that after substitution it becomes a subtype of  $F_{right}$  as defined by [Function Type Conformance Non-Parameterized](#))
- else if  $|F_{left.typePars}| = |F_{right.typePars}|$ :

- $\Gamma \leftarrow \{V_i^r \leftarrow V_i^l | 0 \leq i \leq n\} \vdash Fleft < :Fright$  ( accordingly)

- -

$$\forall 0 \leq i \leq n:$$

$$\text{intersection}\{V_i^l.\text{upperBounds}\} :> \text{intersection}\{V_i^r.\text{upperBounds}\}$$

with  $Fleft.typePars = V_0^l, V_1^l, \dots, V_n^l$  and  $Fright.typePars = V_0^r, V_1^r, \dots, V_n^r$

(i.e. we replace each type variable in  $Fright$  by the corresponding type variable at the same index in  $Fleft$  and check the constraints from [Function Type Conformance Non-Parameterized](#) as if  $Fleft$  and  $Fright$  were non-parameterized functions and, in addition, the upper bounds on the left side need to be supertypes of the upper bounds on the right side).

Note that the upper bounds on the left must be supertypes of the right-side upper bounds (for similar reasons why types of formal parameters on the left are required to be supertypes of the formal parameters' types in ). Where a particular type variable is used, on co- or contra-variant position, is not relevant:

#### Example 6.4. Bounded type variable at co-variant position in function type

```
class A {}
class B extends A {}

class X {
    <T extends B> m(): T { return null; }
}
class Y extends X {
    @Override
    <T extends A> m(): T { return null; }
}
```

Method `m` in `Y` may return an `A`, thus breaking the contract of `m` in `X`, but only if it is parameterized to do so, which is not allowed for clients of `X`, only those of `Y`. Therefore, the override in the above example is valid.

The subtype relation for function types is also applied for method overriding to ensure that an overriding method's signature conforms to that of the overridden method, see [\[Req-IDE-72\]](#) (applies to method consumption and implementation accordingly, see [\[Req-IDE-73\]](#) and [\[Req-IDE-74\]](#)). Note that this is very different from Java which is far more restrictive when checking overriding methods. As Java also supports method overloading: given two types  $A, B$  with  $B < :A$  and a super class method `void m(B param)`, it is valid to override `m` as `void m(A param)` in N4JS but not in Java. In Java this would be handled as method overloading and therefore an `@Override` annotation on `m` would produce an error.

#### Upper and Lower Bound of a Function Type

##### Requirement: IDE-80: Upper and Lower Bound of a Function Type (ver. 1)

The upper bound of a function type  $F$  is a function type with the lower bound types of the parameters and the upper bound of the return type:

$\text{upper(function}(P_1, \dots, P_n) : R) := \text{function(lower}(P_1), \dots, \text{lower}(P_n)) : \text{upper}(R)$

The lower bound of a function type  $F$  is a function type with the upper bound types of the parameters and the lower bound of the return type:

$\text{lower(function}(P_1, \dots, P_n) : R) := \text{function(upper}(P_1), \dots, \text{upper}(P_n)) : \text{lower}(R)$

### 6.1.3. Autoboxing of Function Type

Function types, compared to other types like String, come only in one flavour: the Function object representation. There is no primitive function type. Nevertheless, for function type expressions and function declarations, it is possible to call the properties of Function object directly. This is similar to autoboxing for strings.

#### Access of Function properties on functions

```
// function declaration
var param: number = function(a,b){}.length // 2
```

```

function a(x: number) : number { return x*x; }
// function reference
a.length; // 1

// function variable
var f = function(m,l,b){/*...*/};
f.length; // 3

class A {
  s: string;
  sayS(): string{ return this.s; }
}

var objA: A = new A();
objA.s = "A";

var objB = {s:"B"}

// function variable
var m = objA.sayS; // method as function, detached from objA
var mA: {function(any)} = m.bind(objA); // bind to objA
var mB: {function(any)} = m.bind(objB); // bind to objB

m() // returns: undefined
mA() // returns: A
mB() // returns: B

m.call(objA,1,2,3); // returns: A
m.apply(objB,[1,2,3]); // returns: B
m.toString(); // returns: function sayS(){ return this.s; }

```

## 6.1.4. Arguments Object

A special arguments object is defined within the body of a function. It is accessible through the implicitly-defined local variable named `arguments`, unless it is shadowed by a local variable, a formal parameter or a function named `arguments` or in the rare case that the function itself is called 'arguments' [ECMA11a(p.S10.5, p.p.59)]. The argument object has array-like behavior even though it is not of type `array`:

- All actual passed-in parameters of the current execution context can be retrieved by 0-based index access.
- The `length` property of the arguments object stores the actual number of passed-in arguments which may differ from the number of formally defined number of parameters *fpars* of the containing function.
- It is possible to store custom values in the arguments object, even outside the original index boundaries.
- All obtained values from the arguments object are of type `any`.

In non-strict ES mode the `callee` property holds a reference to the function executed [ECMA11a(p.S10.6, p.p.61)].

Arguments.callee

**Requirement: IDE-81: Arguments.callee (ver. 1)**

In N4JS and in ES strict mode the use of `arguments.callee` is prohibited.

Arguments as formal parameter name

**Requirement: IDE-82: Arguments as formal parameter name (ver. 1)**

In N4JS, the formal parameters of the function cannot be named `arguments`. This applies to all variable execution environments like field accessors (getter/setter, Section 5.2.6.4, "Field Accessors (Getter/Setter)", methods (Section 5.2.3, "Methods") and constructors (Section 4.11, "Constructor and Classifier Type"), where `FormalParameter` type is used.

```

// regular function
function a1(s1: string, n2: number) {
  var l: number = arguments.length;
  var s: string = arguments[0] as string;
}

class A {
  // property access
  get s(): string { return ""+arguments.length; } // 0
  set s(n: number) { console.log( arguments.length ); } // 1

```

```

// method
m(arg: string) {
    var l: number = arguments.length;
    var s: string = arguments[0] as string;
}
}

// property access in object literals
var x = {
    a:5,
    get b(): string {
        return ""+arguments.length
    }
}

// invalid:
function z(){
    arguments.length // illegal, see next lines
    // define arguments to be a plain variable of type number:
    var arguments: number = 4;
}

```

## 6.2. ECMAScript 5 Function Definition

### 6.2.1. Function Declaration

#### 6.2.1.1. Syntax

A function can be defined as described in [ECMA11a(p.S13, p.p.98)] and additional annotations can be specified. Since N4JS is based on [ECMA15a], the syntax contains constructs not available in [ECMA11a]. The newer constructs defined only in [ECMA15a] and proposals already implemented in N4JS are described in Section 6.3, “ECMAScript 2015 Function Definition” and Section 6.4, “ECMAScript Proposals Function Definition”.



In contrast to plain JavaScript, function declarations can be used in blocks in N4JS. This is only true, however, for N4JS files, not for plain JS files.

#### Syntax Function Declaration and Expression

```

FunctionDeclaration <Yield>:
    => ( {FunctionDeclaration}
        annotations+=Annotation*
        (declaredModifiers+=N4Modifier)*
        -> FunctionImpl <Yield,Yield,Expression=false>
    ) => Semi?
;

fragment AsyncNoTrailingLineBreak *: (declaredAsync?='async' NoLineTerminator)?;

fragment FunctionImpl<Yield, YieldIfGenerator, Expression>*:
    'function'
    (
        generator?='*' FunctionHeader<YieldIfGenerator,Generator=true> FunctionBody<Yield=true,Expression>
        | FunctionHeader<Yield,Generator=false> FunctionBody<Yield=false,Expression>
    )
;

fragment FunctionHeader<Yield, Generator>*:
    TypeVariables?
    name=BindingIdentifier<Yield>?
    StrictFormalParameters<Yield=Generator>
    (-> !' returnTypeRef=TypeRef)?
;

fragment FunctionBody <Yield, Expression>*:
    <Expression> body=Block<Yield>
    | <!Expression> body=Block<Yield>?
;
```

Properties of the function declaration and expression are described in Section 6.1, “Function Type”.

For this specification, we introduce a supertype *FunctionDefinition* for both, *FunctionDeclaration* and *FunctionExpression*. This supertype contains all common properties of these two subtypes, that is, all properties of *FunctionExpression*.

**Example 6.5. Function Declaration with Type Annotation**

```
// plain JS
function f(p) { return p.length }
// N4JS
function f(p: string): number { return p.length }
```

### 6.2.1.2. Semantics

A function defined in a class's method (or method modifier) builder is a method, see [Section 5.2.3, “Methods”](#) for details and additional constraints. The metatype of a function definition is function type ([Section 6.1, “Function Type”](#)), as a function declaration is only a different syntax for creating a `Function` object. Constraints for function type are described in [Section 6.1, “Function Type”](#). Another consequence is that the inferred type of a function definition  $f_{decl}$  is simply its function type  $F$ .

$[f_{decl}]$   
 $[F]$

Note that the type of a function definition is different from its return type  $f_{decl}$ !

**Function Declaration only on Top-Level****Requirement: IDE-83: Function Declaration only on Top-Level (ver. 1)**

1. In plain JavaScript, function declarations must only be located on top-level, that is they must not be nested in blocks. Since this is supported by most JavaScript engines, only a warning is issued.

### 6.2.2. Function Expression

A function expression [[ECMA11a\(p.S11.2.5\)](#)] is quite similar to a function declaration. Thus, most details are explained in [Section 6.2, “ECMAScript 5 Function Definition”](#).

#### 6.2.2.1. Syntax

```
FunctionExpression:
  ( {FunctionExpression}
    FunctionImpl<Yield=false,YieldIfGenerator=true,Expression=true>
  )
;
```

#### 6.2.2.2. Semantics and Type Inference

In general, the inferred type of a function expression simply is the function type as described in [Section 6.1, “Function Type”](#). Often, the signature of a function expression is not explicitly specified but it can be inferred from the context. The following context information is used to infer the full signature:

- If the function expression is used on the right hand side of an assignment, the expected return type can be inferred from the left hand side.
- If the function expression is used as an argument in a call to another function, the full signature can be inferred from the corresponding type of the formal parameter declaration.

Although the signature of the function expression may be inferred from the formal parameter if the function expression is used as argument, this inference has some conceptual limitations. This is demonstrated in the next example.

**Example 6.6. Inference Of Function Expression’s Signature**

In general, `{function():any}` is a subtype of `{function():void}` (cf. [Section 6.1, “Function Type”](#)). When the return type of a function expression is inferred, this relation is taken into account which may lead to unexpected results as shown in the following code snippet:

```
function f(cb: {function():void}) { cb() }
f(function() { return 1; })
```

No error is issued: The type of the function expression actually is inferred to `{function():any}`, because there is a return statement with an expression. It is not inferred to `{function():void}`, even if the formal parameter of `f` suggests that.

Due to the previously-stated relation `{function():any} <: {function():void}` this is correct – the client (in this case function `f`) works perfectly well even if `cb` returns something. The contract of arguments states that the type of the argument is a subtype of the type of the formal parameter. This is what the inferencer takes into account!

## 6.3. ECMAScript 2015 Function Definition

### 6.3.1. Formal Parameters

Parameter handling has been significantly upgraded in ECMAScript 6. It now supports parameter default values, rest parameters (variadics) and destructuring. Formal parameters can be modified to be either default or variadic. In case a formal parameter has no modifier, it is called normal. Modified parameters also become optional.

Modifiers of formal parameters such as default or rest are neither evaluated nor rewritten in the transpiler.

#### 6.3.1.1. Optional Parameters

An optional formal parameter can be omitted when calling a function/method. An omitted parameter has the value `undefined`. In case the omitted parameter is variadic, the value is an empty array.

Parameters can not be declared as optional explicitly. Instead, being optional is true when a parameter is declared as default or variadic. Note that any formal parameter that follows a default parameter is itself also a default thus an optional parameter.

#### 6.3.1.2. Default Parameters

A default parameter value is specified for a parameter via an equals sign (`=`). If a caller doesn't provide a value for the parameter, the default value is used.

Default initializers of parameters are specified at a formal parameter of a function or method after the equal sign using an arbitrary initializer expression, such as `var = "s"`. However, this default initializer can be omitted. When a formal parameter has a declared type, the default initializer is specified at the end, such as: `var : string = "s"`. The initializer expression is only evaluated in case no actual argument is given for the formal parameter. Also, the initializer expression is evaluated when the actual argument value is `undefined`.

Formal parameters become default parameters implicitly when they are preceded by an explicit default parameter. In such cases, the default initializer is `undefined`.

#### Default parameters

**Requirement: IDE-14501: Default parameters** (ver. 1)

Any normal parameter which is preceded by a default parameter also becomes a default parameter. Its initializer is `undefined`.

When a method is overwritten, its default parameters are not part of the overwriting method. Consequently, initializers of default parameters in abstract methods are obsolete.

#### 6.3.1.3. Variadic

Variadic parameters are also called *rest parameters*. Marking a parameter as variadic indicates that method accepts a variable number of parameters. A variadic parameter implies that the parameter is also optional as the cardinality is defined as `[0..*]`. No further parameter can be defined after a variadic parameter. When no argument is given for a variadic parameter, an empty array is provided when using the parameter in the body of the function or method.

#### Variadic and optional parameters

**Requirement: IDE-16: Variadic and optional parameters** (ver. 1)

For a parameter `p`, the following condition must hold: `p.var → p.opt`.

A parameter can not be declared both variadic and with a default value. That is to say that one can either write `varName = (default)` or `... varName`, but not `... varName =`.

Declaring a variadic parameter of type  $T$  causes the type of the method parameter to become `Array<T>`. That is, declaring `function(...tags : string)` causes `tags` to be an `Array<string>` and not just a scalar `string` value.

To make this work at runtime, the compiler will generate code that constructs the `parameter` from the `arguments` parameter explicitly passed to the function.

#### Variadic at Runtime

**Requirement: IDE-17: Variadic at Runtime (ver. 1)**

At runtime, a variadic parameter is never set to undefined. Instead, the array may be empty. This must be true even if preceding parameters are optional and no arguments are passed at runtime.

For more constraints on using the variadic modifier, see [Section 4.7.3, “Function-Object-Type”](#).

### 6.3.2. Generator Functions

Generators come together with the `yield` expression and can play three roles: the role of an iterator (data producer), of an observer (data consumer), and a combined role which is called coroutines. When calling a generator function or method, the returned generator object of type `Generator<TYield,TReturn,TNext>` can be controlled by its methods (cf. [[ECMA15a\(p.S14.4\)](#)], also see [[Kuizinas14a](#)]).

#### 6.3.2.1. Syntax

Generator functions and methods differ from ordinary functions and methods only in the additional `*` symbol before the function or method name. The following syntax rules are extracted from the real syntax rules. They only display parts relevant to declaring a function or method as a generator.

```
GeneratorFunctionDeclaration <Yield>:
  (declaredModifiers+=N4Modifier)*
  'function' generator?= '*'*
  FunctionHeader<YieldIfGenerator,Generator=true>
  FunctionBody<Yield=true,Expression=false>
;

GeneratorFunctionExpression:
  'function' generator?= '*'
  FunctionHeader<YieldIfGenerator,Generator=true>
  FunctionBody<Yield=true,Expression=true>
;

GeneratorMethodDeclaration:
  annotations+=Annotation+ (declaredModifiers+=N4Modifier)* TypeVariables?
  generator?= '*' NoLineTerminator LiteralOrComputedPropertyName<Yield>
  MethodParamsReturnAndBody<Generator=true>
```

#### 6.3.2.2. Semantics

The basic idea is to make code dealing with Generators easier to write and more readable without changing their functionality. Take this example:

##### Example 6.7. Two simple generator functions

```
// explicit form of the return type
function * countTo(iMax:int) : Generator<int,string,undefined> {
  for (int i=0; i<=iMax; i++)
    yield i;
  return "finished";
}
val genObj1 = countTo(3);
val values1 = [...genObj1]; // is [0,1,2,3]
val lastObj1 = genObj1.next(); // is {value="finished",done=true}

// shorthand form of the return type
function * countFrom(start:int) : int {
  for (int i=start; i>=0; i--)
    yield i;
  return finished;
}
```

```

val genObj2 = countFrom(3);
val values2 = [...genObj2]; // is [3,2,1,0]
val lastObj2 = genObj2.next(); // is {value="finished",done=true}

```

In the example above, two generator functions are declared. The first declares its return type explicitly whereas the second uses a shorthand form.

Generator functions and methods return objects of the type `Generator<TYield,TReturn,TNext>` which is a subtype of the `Iterable<TYield>` and `Iterator<TYield>` interfaces. Moreover, it provides the methods `throw(exception:any)` and `return(value:TNext?)` for advanced control of the generator object. The complete interface of the generator class is given below.

### The generator class

```

public providedByRuntime interface Generator<out TYield, out TReturn, in TNext>
extends Iterable<TYield>, Iterator<TYield> {
public abstract next(value: TNext?): IteratorEntry<TYield>
public abstract [Symbol.iterator](): Generator<TYield, TReturn, TNext>
public abstract throw(exception: any): IteratorEntry<TYield>;
public abstract return(value: TNext?): IteratorEntry<TReturn>;
}

```

Modifier `*`

#### Requirement: IDE-14370: Modifier `pass:[\*]` (ver. 1)

. `*` may be used on declared functions and methods, and for function expressions. . A function or method `f` with a declared return type `R` that is declared `*` has an actual return type of `Generator<TYield,TReturn,TNext>` . . A generator function or method can have no declared return type, a shorthand form of a return type or an explicitly declared return type. . . The explicitly declared return type is of the form `Generator<TYield,TReturn,TNext>` with the type variables: ... `TYield` as the expected type of the yield expression argument, ... `TReturn` as the expected type of the return expression, and ... `TNext` as both the return type of the yield expression. . . The shorthand form only declares the type of `TYield` which implicitly translates to `Generator<TYield,TReturn,any>` as the return type. . . The type `TReturn` is inferred to either `undefined` or `any` from the body. . . In case the declared type is `void`, actual return type evaluates to `Generator<undefined,undefined,any>` . . If no return type is declared, both `TYield` and `TReturn` are inferred from the body to either `any` or `undefined`. `TNext` is `any` . . Given a generator function or method `f` with an actual return type `Generator<TYield,TReturn,TNext>` : . . all yield statements in `f` must have an expression of type `TYield` . . all return statements in `f` must have an expression of type `TReturn` . . Return statements in generator functions or methods are always optional.

Modifier `yield` and `yield*`

#### Requirement: IDE-14371: Modifier `yield` and `yield\*` (ver. 1)

. `yield` and `yield*` may only be in body of generator functions or methods. . `yield expr` takes only expressions `expr` of type `TYield` in a generator function or methods with the actual type `Generator<TYield,TReturn,TNext>` . . The return type of the `yield` expression is `TNext`. . `yield* fg()` takes only iterators of type `Iterator<TYield>`, and generator functions or methods `fg` with the actual return type `Generator<?,? extends TYield,? extends TReturn,? super TNext>` . . The return type of the `yield*` expression is `any`, since a custom iterator could return an entry `{done=true,value}` and any value for the variable `value`.

Similar to `async` functions, shorthand and explicit form `* function():int{};` and `* function():Generator<int,TResult,any>` are equal, given that the inferred `TResult` of the former functions equals to `TResult` in the latter function). In other words, the return type of generator functions or methods is wrapped when it is not explicitly defined as `Generator` already. Thus, whenever a nested generator type is desired, it has to be defined explicitly. Consider the example below.

### Type variables with `async` methods.

```

class C<T> {
genFoo(): T{} // equals to genFoo(): Generator<T, undefined, any>;
  // note that TResult depends on the body of genFoo()
}
function fn(C<int> c1, C<Generator<int,any,any>> c2) {
  c1.genFoo(); // returns Generator<int, undefined, any>
  c2.genFoo(); // returns Generator<Generator<int,any,any>, undefined, any>
}

```

### 6.3.2.3. Generator Arrow Functions

As of now, generator arrow functions are not supported by EcmaScript 6 and also, the support is not planned. However, introducing generator arrow function in EcmaScript is still under discussion. For more information, please refer to [ESDiscuss.org](#) and [StackOverflow.com](#).

### 6.3.3. Arrow Function Expression

This is an ECMAScript 6 expression (see [\[ECMA15a\(p.S14.2\)\]](#)) for simplifying the definition of anonymous function expressions, a.k.a. lambdas or closures. The ECMAScript Specification calls this a function definition even though they may only appear in the context of expressions.

Along with Assignments, Arrow function expressions have the least precedence, e.g. they serve as the entry point for the expression tree.

Arrow function expressions can be considered syntactic window-dressing for old-school function expressions and therefore do not support the benefits regarding parameter annotations although parameter types may be given explicitly. The return type can be given as type hint if desired, but this is not mandatory (if left out, the return type is inferred). The notation `@=>` stands for an async arrow function ([Section 6.4.1.3, “Asynchronous Arrow Functions”](#)).

#### 6.3.3.1. Syntax

The simplified syntax reads like this:

```

ArrowExpression returns ArrowFunction:
  =>(
    {ArrowFunction}
    (
      '('
        ( fpars+=FormalParameterNoAnnotations ( ',' fpars+=FormalParameterNoAnnotations )* )?
      ')'
      ('@' returnTypeRef=TypeRef)?
    | fpars+=FormalParameterNoType
    )
    '>>'
  ) (
    (=> hasBracesAroundBody?='{' body=BlockMinusBraces '}') | body=ExpressionDisguisedAsBlock
  )
;

FormalParameterNoAnnotations returns FormalParameter:
  (declaredTypeRef=TypeRef variadic?='...')? name=JSIdentifier
;
FormalParameterNoType returns FormalParameter: name=JSIdentifier;

BlockMinusBraces returns Block: {Block} statements+=Statement*;

ExpressionDisguisedAsBlock returns Block:
  {Block} statements+=AssignmentExpressionStatement
;

AssignmentExpressionStatement returns ExpressionStatement: expression=AssignmentExpression;

```

#### 6.3.3.2. Semantics and Type Inference

Generally speaking, the semantics are very similar to the function expressions but the devil's in the details:

- `arguments`: Unlike normal function expressions, an arrow function does not introduce an implicit `arguments` variable ([Section 6.1.4, “Arguments Object”](#)), therefore any occurrence of it in the arrow function's body has always the same binding as an occurrence of `arguments` in the lexical context enclosing the arrow function.
- `this`: An arrow function does not introduce a binding of its own for the `this` keyword. That explains why uses in the body of arrow function have the same meaning as occurrences in the enclosing lexical scope. As a consequence, an arrow function at the top level has both usages of `arguments` and `this` flagged as error (the outer lexical context doesn't provide definitions for them).
- `super`: As with function expressions in general, whether of the arrow variety or not, the usage of `super` isn't allowed in the body of arrow functions.

No This in Top Level Arrow Function in N4JS Mode

**Requirement: IDE-84: No This in Top Level Arrow Function in N4JS Mode (ver. 1)**

In N4JS, a top-level arrow function can't refer to `this` as there's no outer lexical context that provides a binding for it.

### No Arguments in Top Level Arrow Function

**Requirement: IDE-85: No Arguments in Top Level Arrow Function (ver. 1)**

In N4JS, a top-level arrow function can't include usages of `arguments` in its body, again because of the missing binding for it.

## 6.4. ECMAScript Proposals Function Definition

### 6.4.1. Asynchronous Functions

To improve language-level support for asynchronous code, there exists an ECMAScript proposal<sup>1</sup> based on Promises which are provided by ES6 as built-in types. N4JS implements this proposal. This concept is supported for declared functions and methods (Section 5.2.4.1, “Asynchronous Methods”) as well as for function expressions and arrow functions (Section 6.4.1.3, “Asynchronous Arrow Functions”).

#### 6.4.1.1. Syntax

The following syntax rules are extracted from the real syntax rules. They only display parts relevant to declaring a function or method as asynchronous.

```

AsyncFunctionDeclaration <Yield>:
  (declaredModifiers+=N4Modifier)*
  declaredAsync?= 'async' NoLineTerminator 'function'
  FunctionHeader<Yield,Generator=false>
  FunctionBody<Yield=false,Expression=false>
;

AsyncFunctionExpression:
  declaredAsync?= 'async' NoLineTerminator 'function'
  FunctionHeader<Yield=false,Generator=false>
  FunctionBody<Yield=false,Expression=true>
;

AsyncArrowExpression <In, Yield>:
  declaredAsync?= 'async' NoLineTerminator '('
  (fpars+=FormalParameter<Yield>
   (',' fpars+=FormalParameter<Yield>)*)?
  ')' (':' returnTypeRef=TypeRef)? '=>'
  ( '{' body=BlockMinusBraces<Yield> '}'
  | body=ExpressionDisguisedAsBlock<In>
  )
;

AsyncMethodDeclaration:
  annotations+=Annotation+ (declaredModifiers+=N4Modifier)* TypeVariables?
  declaredAsync?= 'async' NoLineTerminator LiteralOrComputedPropertyName<Yield>
  MethodParamsReturnAndBody
;
```

‘async’ is not a reserved word in ECMAScript and it can therefore be used either as an identifier or as a keyword, depending on the context. When used as a modifier to declare a function as asynchronous, then there must be no line terminator after the `async` modifier. This enables the parser to distinguish between using `async` as an identifier reference and a keyword, as shown in the next example.

#### Example 6.8. Async as keyword and identifier

```

1  async
2  function foo() {}
3  // vs
4  async function bar(); 2
;
```

- 1** In this snippet, the `async` on line 1 is an identifier reference (referencing a variable or parameter) and the function defined on line 2 is a non-asynchronous function. The automatic semicolon insertion adds a semicolon after the reference on line 1.
- 2** In contrast, `async` on line 4 is recognized as a modifier declaring the function as asynchronous.

<sup>1</sup> see <http://tc39.github.io/ecmascript-asyncawait/>

### 6.4.1.2. Semantics

The basic idea is to make code dealing with Promises easier to write and more readable without changing the functionality of Promises. Take this example:

#### A simple asynchronous function using `async/await`.

```
// some asynchronous legacy API using promises
interface DB {}
interface DBAccess {
    getDataBase(): Promise<DB,?>
    loadEntry(db: DB, id: string): Promise<string,?>
}

var access: DBAccess;

// our own function using async/await
async function loadAddress(id: string) : string {
    try {
        var db: DB = await access.getDataBase();
        var entry: string = await access.loadEntry(db, id);
        return entry.address;
    }
    catch(err) {
        // either DataBase() or loadEntry() failed
        throw err;
    }
}
```

The modifier `async` changes the return type of `loadAddress()` from `string` (the declared return type) to `Promise<string,?>` (the actual return type). For code inside the function, the return type is still `string`: the value in the return statement of the last line will be wrapped in a Promise. For client code outside the function and in case of recursive invocations, the return type is `Promise<string,?>`. To raise an error, simply throw an exception, its value will become the error value of the returned Promise.

If the expression after an `await` evaluates to a `Promise`, execution of the enclosing asynchronous function will be suspended until either a success value is available (which will then make the entire await-expression evaluate to this success value and continue execution) or until the Promise is rejected (which will then cause an exception to be thrown at the location of the await-expression). If, on the other hand, the expression after an `await` evaluates to a non-promise, the value will be simply passed through. In addition, a warning is shown to indicate the unnecessary `await` expression.

Note how method `loadAddress()` above can be implemented without any explicit references to the built-in type `Promise`. In the above example we handle the errors of the nested asynchronous calls to `getDataBase()` and `loadEntry()` for demonstration purposes only; if we are not interested in the errors we could simply remove the try/catch block and any errors would be forwarded to the caller of `loadAddress()`.

Invoking an `async` function commonly adopts one of two forms:

- `var p: Promise<successType,?> = asyncFn()`
- `await asyncFn()`

These patterns are so common that a warning is available whenever both

1. `Promise` is omitted as expected type; and
2. `await` is also omitted.

The warning aims at hinting about forgetting to wait for the result, while remaining non-noisy.

#### Modifier `async` and `await`

##### Requirement: IDE-86: Modifier `async` and `await` (ver. 1)

1. `async` may be used on declared functions and methods as well as for function expressions and arrow functions.
2. A function or method that is declared `async` can have no declared return type, a shorthand form of a return type or an explicitly declared return type.
  - a. The explicitly declared return type is of the form `Promise<R,E>` where `R` is the type of all return statements in the body, and `E` is the type of exceptions that are thrown in the body.

- b. The shorthand form only declares the type of  $R$  which implicitly translates to `Promise<R,?>` as the actual return type.
  - c. In case no return type is declared, the type  $R$  of `Promise<R,?>` is inferred from the body.
3. A function or method  $f$  with a declared return type  $R$  that is declared `async` has an actual return type of
- a.  $R$  if  $R$  is a subtype of `Promise<?,?>`,
  - b. `Promise<undefined,?>` if  $R$  is type `void`.
  - c. `Promise<R,?>` in all other cases (i.e. the declared return type  $R$  is being wrapped in a `Promise`).
4. Return type inference is only performed when no return type is declared.
- a. The return type  $R$  of `Promise<R,?>` is inferred either as `void` or as `any`.
5. Given a function or method  $f$  that is declared `async` with a declared return type  $R$ , or with a declared return type `Promise<R,?>`, all return statements in  $f$  must have an expression of type  $R$  (and not of type `Promise<R,?>`).
6. `await` can be used in expressions directly enclosed in an `async` function, and behaves like a unary operator with the same precedence as `yield` in ES6.
7. Given an expression  $expr$  of type  $T$ , the type of  $(await expr)$  is inferred to  $T$  if  $T$  is not a `Promise`, or it is inferred to  $S$  if  $T$  is a `Promise` with a success value of type  $S$ , i.e.  $T <: Promise<S,?>$ .

In other words, the return type  $R$  of `async` functions and methods will always be wrapped to `Promise<R,?>` unless  $R$  is a `Promise` already. As a consequence, nested `Promise`s as a return type of a `async` function or method have to be stated explicitly like `Promise<Promise<R,?>,?>`.

When a type variable  $T$  is used to define the the return type of an `async` function or method, it will always be wrapped. Consider the example below.

**Example 6.9. Type variables with `async` methods.**

```
interface I<T> {
  async foo(): T; // amounts to foo(): Promise<T,?>
}

function snafu(i1: I<int>, i2: I<Promise<int,?>>) {
  i1.foo(); // returns Promise<int,?>
  i2.foo(); // returns Promise<Promise<int,?>,?>
}
```

#### 6.4.1.3. Asynchronous Arrow Functions

An `await` expression is allowed in the body of an `async` arrow function but not in the body of a non-`async` arrow function. The semantics here are intentional and are in line with similar constraint for function expressions.

### 6.5. N4JS Extended Function Definition

#### 6.5.1. Generic Functions

A generic function is a function with a list of generic type parameters. These type parameters can be used in the function signature to declare the types of formal parameters and the return type. In addition, the type parameters can be used in the function body, for example when declaring the type of a local variable.

In the following listing, a generic function `foo` is defined that has two type parameters `s` and `T`. Thereby `s` is used as to declare the parameter type `Array<s>` and `T` is used as the return type and to construct the returned value in the function body.

**Generic Function Definition**

```
function <S,T> foo(s: Array<S>): T { return new T(s); }
```

If a generic type parameter is not used as a formal parameter type or the return type, a warning is generated.

#### 6.5.2. Promisifiable Functions

In many existing libraries, which have been developed in pre-ES6-promise-API times, callback methods are used for asynchronous behavior. An asynchronous function follows the following conventions:

```
'function' name '(' arbitraryParameters ',' callbackFunction ')'
```

Usually the function returns nothing (`void`). The callback function usually takes two arguments, in which the first is an error object and the other is the result value of the asynchronous operation. The callback function is called from the asynchronous function, leading to nested function calls (aka 'callback hell').

In order to simplify usage of this pattern, it is possible to mark such a function or method as `@Promisifiable`. It is then possible to 'promisify' an invocation of this function or method, which means no callback function argument has to be provided and a will be returned. The function or method can then be used as if it were declared with `async`. This is particularly useful in N4JS definition files (.n4jsd) to allow using an existing callback-based API from N4JS code with the more convenient `await`.

#### Example 6.10. Promisifiable

Given a function with an N4JS signature

```
f(x: int, cb: {function(Error, string)}): void
```

This method can be annotated with `Promisifiable` as follows:

```
@Promisifiable f(x: int, cb: {function(Error, string)}): void
```

With this annotation, the function can be invoked in four different ways:

```
f(42, function(err, result1) { /* ... */});           // traditional
var promise: Promise<string, Error> = @Promisify f(42); // promise
var result3: string = await @Promisify f(42);          // long
var result4: string = await f(42);                      // short
```

The first line is only provided for completeness and shows that a promisifiable function can still be used in the ordinary way by providing a callback - no special handling will occur in this case. The second line shows how `f` can be promisified using the `@Promisify` annotation - no callback needs to be provided and instead, a `Promise` will be returned. We can either use this promise directly or immediately `await` on it, as shown in line 3. The syntax shown in line 4 is merely shorthand for `await @Promisify`, i.e. the annotation is optional after `await`.

#### Promisifiable

##### Requirement: IDE-87: Promisifiable (ver. 1)

A function or method  $f$  can be annotated with `@Promisifiable` if and only if the following constraints hold:

1. Last parameter of  $f$  is a function (the *callback*).

2. The *callback* has a signature of

- `{function(E, T0, T1, ..., Tn): V}`, or
- `{function(T0, T1, ..., Tn): V}`

in which  $E$  is type `Error` or a subtype thereof,  $T_0, \dots, T_n$  are arbitrary types except or its subtypes.  $E$ , if given, is then the type of the error value, and  $T_0, \dots, T_n$  are the types of the success values of the asynchronous operation.

Since the return value of the synchronous function call is not available when using `@Promisify`,  $V$  is recommended to be `void`, but it can be any type.

3. The *callback* parameter may be optional.<sup>2</sup>

According to [Req-IDE-87], a promisifiable function or method may or may not have a non-void return type, and that only the first parameter of the *callback* is allowed to be of type `Error`, all other parameters must be of other types.

#### @Promisify and await with promisifiable functions

##### Requirement: IDE-88: @Promisify and await with promisifiable functions (ver. 1)

A promisifiable function  $f$  with one of the two valid signatures given in [Req-IDE-87] can be promisified with `Promisify` or used with `await`, if and only if the following constraints hold:

1. Function  $f$  must be annotated with `@Promisifiable`.

2. Using `@Promisify f()` without `await` returns a promise of type `Promise<S, F>` where

<sup>2</sup> Even in this case, the function will actually be called with the callback method which is then created by the transpiler. However, the callback is not given in the N4JS code).

- $S$  is `IterableN<T0, ..., Tn>` if  $n \geq 2$ , `T` if  $n = 1$ , and `undefined` if  $n = 0$ .
  - $F$  is `E` if given, `undefined` otherwise.
3. Using `await @Promisify f()` returns a value of type `IterableN<T0, ..., Tn>` if  $n \geq 2$ , of type `T` if  $n = 1$ , and of type `undefined` if  $n = 0$ .
  4. In case of using an `await`, the annotation can be omitted.  
I.e., `await @Promisify f()` is equivalent to `await f()`.
  5. Only call expressions using  $f$  as target can be promisified, in other words this is illegal:

```
var pf = @Promisify f; // illegal code!
```



# Chapter 7. Conversions and Reflection

## 7.1. Autoboxing and Coercing

Coercing is the ability to implicitly cast one (primitive) type to another. Autoboxing is a special kind of coercing in that is the ability to automatically convert a primitive value type, such as `string`, `number`, or `boolean`, to its corresponding Object type version `String`, `Number`, `Boolean`. The capital letters in the latter are an essential distinction.

Conversion between primitives and object-representations of a datatype are not automatic in N4JS. Only in the cases of object-method invocations on a primitive type (for `string` to call `"abc".length`, for example) automatic conversion is applied.

Note that N4JS specific primitive types `pathselector` and `i18nkey` are handled similarly to `string`.

### 7.1.1. Coercing

In [ECMA11a], coercing is defined by means of the abstract specification method `ToPrimitive` [ECMA11a(p.S9.1)], also see [ECMA11a(p.S9.10)]. Other conversions, such as `ToNumber`, are not directly supported but reflected in the typing rules of expressions.

We express absence of automatic coercion here by means of subtype relations:

$$\begin{array}{ll} \overline{\Gamma \vdash \text{Boolean} \not\leq \text{boolean}} & \overline{\Gamma \vdash \text{boolean} \not\leq \text{Boolean}} \\ \overline{\Gamma \vdash \text{Number} \not\leq \text{number}} & \overline{\Gamma \vdash \text{number} \not\leq \text{Number}} \\ \overline{\Gamma \vdash \text{String} \not\leq \text{string}} & \overline{\Gamma \vdash \text{string} \not\leq \text{String}} \end{array}$$

and for the N4JS specific types:

$$\begin{array}{l} \overline{\Gamma \vdash \text{pathSelector}\langle T \rangle \leq \text{string}} \\ \overline{\Gamma \vdash \text{i18nKey} \leq \text{string}} \end{array}$$

If a conversion between primitive and object type is desired, we require the user of N4JS to actively convert the values. The reason for that is the notably different behavior of object- and primitive-variants of a type in expression evaluation:

```
var bool: boolean = false;
var Bool: Boolean = new Boolean( false );

console.log( bool ? "true" : "false" ); // prints "false"
console.log( Bool ? "true" : "false" ); // prints "true"!
```

Conversion between a primitive type to its object-variant is achieved by the `new` operator. The `valueOf()` method converts the object-variant back to a primitive value.

```
// objects from literals:
var bo: Boolean = new Boolean( true ); // typeof bo: object
var no: Number = new Number( 42 ); // typeof no: object
var so: String = new String( "foo" ); // typeof so: object

// to primitive
var b: boolean = bo.valueOf(); // typeof b: boolean -- true
var n: number = no.valueOf(); // typeof n: number -- 42
var s: string = so.valueOf(); // typeof s: string -- "foo"

// to object-type
bo = new Boolean( b );
no = new Number( n );
so = new String( s );
```

Conversion of variables of type `Object` or from one primitive type to another is expressed in terms of typing rules for expressions. That is, it is not possible to convert any `Object` to a primitive in general, but it is possible to do so in the context of certain expressions such as additive operator. The applied conversions are described in [Section 7.2.1, “Auto-Conversion of Class Instances”](#)

### 7.1.2. Autoboxing of Primitives

In [ECMA11a], autoboxing is defined by `ToObject` [ECMA11a(p.S9.9)].

Autoboxing is not directly supported in N4JS. Instead, primitive types virtually have the same members as their corresponding object types. It is then possible to use the Autoboxing feature when calling a member. In general, Autoboxing is only supported for accessing built-in read-only (immutable) properties. For example, `"some string value".split(" ")`; is supported but `"some string value".foo=1;` will be rejected as String does not allow properties to be added (cf. `String` vs. `String+`, see [Section 4.9.1, "Dynamic"](#)).

Autoboxing often leads to problems, in particular in combination with dynamic types – this is why it is not directly supported in N4JS.

```
var s: String+ = "Hello"; ①
s.prop = 1;
console.log(s.prop); ②
```

- ① will produce an error to prevent the following scenario:
- ② prints "undefined"

### 7.1.3. Autoboxing of Function Expressions and Declarations

Function expressions and declarations always define an object of type `Function`, thus coercing or Autoboxing is not required in case of functions:

It is always possible to use a function expression where a `Function` is required, and to use an object of type `Function` where a function expression is expected. This is only possible if the function signatures are subtype-compatible, see [Section 6.1, "Function Type"](#) for details.

Still, it is always possible to call members of `Function`, e.g., `function(){}.length()`.

## 7.2. Auto-Conversion of Objects

### 7.2.1. Auto-Conversion of Class Instances

All classes defined in N4JS modules implicitly subclass `N4Object`, which is a plain JavaScript Object type. The same auto-conversion rules defined for JavaScript `Object` therefore apply to `N4Object` instances as well.

The basic conversion uses the abstract JavaScript function `ToPrimitive` [[ECMA11a\(p.S9.1\)](#)], which relays on the specification method `Object` [[ECMA11a\(p.S8.12.8\)](#)]. `DefaultValue` calls `valueOf` or `toString` methods if they are defined by the class (in the `methods`-builder).

Note that according to the [[ECMA11a](#)], in most cases, objects are first converted into primitives. That is, in most cases, no extra hint is passed to `DefaultValue`. Thus `valueOf` usually takes precedence over `toString` as demonstrated in the following example:

#### Example 7.1. Auto-Conversion

Assume some classes and corresponding instances defined as follows:

```
class A {}
class B{
    @Override public toString(): string { return "MyB" }
}
class C{
    @Override public valueOf(): any { return 10 }
}
class D{
    @Override public toString(): string { return "MyD" }
    @Override public valueOf(): any { return 20 }
}
var a = new A(), b = new B(), c = new C(), d = new D();
```

Instances of these classes will be converted as demonstrated as follows:

```
console.log(a+"");           // [object Object]
console.log(a+1);            // [object Object]1

console.log(" "+b+" ");
console.log(1+b+1);          // MyB
                           // 1MyB1

console.log(c+"");
console.log(c+1);             // 10
                           // 11
```

```
console.log(d+" ");
// 20
console.log(d+1);
// 21
```

### 7.2.1.1. Auto-Conversion of Interface Instances

Instances of interfaces actually are instances of classes at runtime. The auto-conversion rules described in [Section 7.2.1, “Auto-Conversion of Class Instances”](#) are applied to instances declared as instances of interfaces as well.

## 7.2.2. Auto-Conversion of Enum Literals

Enumeration values are objects and thus follow the behavior for ECMAScript `Object` and `Function`. They have a custom `toString` method which returns the name of the enumeration value.

## 7.3. Type Cast and Type Check

### 7.3.1. Type Cast

(IDEBUG-56): Casting to TypeVars

Type casts are expressed with the cast expression (`as`), see [Section 8.4.2, “Cast \(As\) Expression”](#) for details.

We first define helper rules for the type cast constraints as follows:

$$\begin{aligned}
 & \mu(T) \in \{\text{TEnum, Class, Primitive, ObjectType}\} \\
 & \quad \text{isCPOE}(T) \\
 \underline{\mu(T) \in \{\text{ClassifierType, TypeType}\} \wedge \mu(T.typeRef) \neq \text{TypeVariable}} \\
 & \quad \text{isCPOE}(T) \\
 & \quad \underline{\mu(T) \in \{\text{Interface}\}} \\
 & \quad \text{isInterface}(T) \\
 & \underline{\mu(T) = \text{TypeVariable} \wedge T.upperBounds \neq \emptyset} \\
 & \quad \text{isBoundTypeVar}(T) \\
 \underline{\mu(T) \in \{\text{TEnum, Primitive}\} \vee (\mu(T) = \text{Class} \wedge T.extensibility = \text{final})} \\
 & \quad \text{isFinalByType}(T)
 \end{aligned}$$

#### Cast Validation At Compile Time

**Requirement: IDE-89: Cast Validation At Compile Time (ver. 1)**

Given a type cast expression `e` in which  $\Gamma \vdash e.expr:S$  and target type `T`, the following constraints must hold:

1. `T` must be a classifier, enum, primitive, function type expression, classifier type, type variable, union or intersection type:

$$\mu(T) \in \{\text{any, Class, Interface, Enum, Primitive, ObjectType, FunctionTypeExpression, ClassifierType, TypeVariable, Union, Intersection}\}$$

2. if `s` is a subtype of `T`, the cast is unnecessary and a warning will be generated.
3. if `s` and `T` are classes, enums or primitive types, then `T` must be a subtype of `s`. This is also true if `T` is an interface and the type of `s` cannot have subtypes, or vice versa.

$$\begin{aligned}
 & (\text{isCPOE}(T) \\
 & \quad \wedge (\text{isCPOE}(S) \\
 & \quad \vee (\mu(S) = \text{Intersection} \wedge \exists S' \in S: \text{isCPOE}(S')) \\
 & \quad \vee (\text{isInterface}(T) \wedge \text{isFinalByType}(S)) \\
 & \quad \vee (\text{isFinalByType}(T) \wedge \text{isInterface}(S))) \rightarrow \Gamma \vdash T < :S
 \end{aligned}$$

4. if `s` is a class, enum or primitive type and `T` is a type-variable, then for each given boundary  $T_i^{up}$  of `T` of type class, enum or primitive `s` must be a member of the type hierarchy: <sup>1</sup>

$$\begin{aligned}
 & (\text{isBoundTypeVar}(T) \wedge \text{isCPOE}(S)) \\
 & \rightarrow \forall T_i^{up} \in T.upperBounds (\text{isCPOE}(T_i^{up}) \rightarrow \Gamma \vdash (T_i^{up} < :S \vee T_i^{up} :> S))
 \end{aligned}$$

5. if `s` is a union or intersection type, then the type cast is valid if it is valid for at least one element of `s`.

<sup>1</sup> `i` iterates over all boundaries

6. if `s` and `T` are generics, and if  $S^0 = T^0$ , a cast is possible if type arguments are sub- or supertypes of each other: <sup>2</sup>

$$\mu(S) = \text{Classifier} \wedge \mu(T) = \text{Classifier} \wedge S^0 = T^0 \rightarrow \\ (\forall S.\text{typeArg}_i < :T.\text{typeArg}_i) \vee (\forall T.\text{typeArg}_i < :S.\text{typeArg}_i)$$

7. If `T` is a union type, then the type cast is valid if it is valid for at least one element of `T`.

8. If `T` is an intersection type, then the type cast is valid if it is valid for all elements of `T`.



`any` is a supertype of all other types, thus it is always possible to cast a variable of type `any` to other (non-composed) types.

### 7.3.2. Type Check

There are basically two ways of testing the type of a variable: `typeof` and `instanceof`. N4JS supports type comparison via the ECMAScript `instanceof` operator. The operator `instanceof` retains its standard ECMAScript behavior (e.g. checking whether a value is an instance of a constructor function), but has additional functionality when used with N4JS types.

When used with an N4JS class, `instanceof` also supports checking against an interface. For N4JS enumeration values, it can be used to check whether the value is part of a specific enumeration.

`typeof` only returns a string with the name of the ECMAScript type, which is `object` for all class instances.

N4JS specific `string` types, that is `pathSelector` and `i18nkey` cannot be tested during runtime. These types, therefore, must not be used in `instanceof` expressions. The same is true for string-based enums and arrays which cannot be tested during runtime, thus string-based enum and array types are not permitted on the right-hand side of `instancesof` constructs. For all types for which the evaluation result of `instanceof` could be computed at compile time, the check is unnecessary and thus it is refused by the compiler. Using structural types on the right-hand side of `instancesof` constructs is also not permitted.

In order to avoid errors at runtime, the `instanceof` operator defines appropriate constraints, see [Section 8.1.16, “Relational Expression”](#) for details.

#### Example 7.2. Type Check Example

Given the following classes and variable:

```
interface I{}
class S{}
class Sub extends S implements I{}

var x = new Sub();
```

`typeof x` will simply return `object`. The following table shows the difference between plain JavaScript `instanceof` and N4JS's `instanceof`:

Check	JavaScript	N4JS
<code>x instanceof Sub</code>	<code>true</code>	<code>true</code>
<code>x instanceof S</code>	<code>true</code>	<code>true</code>
<code>x instanceof I</code>	<code>false</code>	<code>true</code>

## 7.4. Reflection meta-information

All N4JS classes, interfaces and enumerations provide meta-information that is used by the runtime and standard library. All classifiers (including enums) provide meta-information by means of a static getter `n4type`. Since it is static getter, it is actually an instance getter of the constructor (or classifier) of a type, which is the only way to retrieve that information in case of interfaces. For enums, this can be retrieved from instances as well.

This getter is of type `N4Class` which is a built-in type just like `N4Object`. It contains the following members:

`fqn`

The FQN of the type.

<sup>2</sup> `i` iterates over all type args

<code>n4superType</code>	The <code>N4Class</code> of the supertype, may be null if supertype is not an <code>N4Class</code> .
<code>allImplementedInterfaces</code>	List of The <i>FQN</i> of implemented interfaces (transitively but without interfaces implemented by supertype)
<code>get isClass</code>	True if the type is an <code>N4Class</code> .
<code>get isInterface</code>	True if the type is an <code>N4Interface</code> .

### 7.4.1. Reflection for Classes

The meta-information for classes is available by means of `N4Object`'s static getter `n4type`. Since it is static getter, it is actually an instance getter of the constructor of a type.

In addition, the static method `of` in `N4Type` is available to retrieve the meta-information for a given instance or constructor. This also allows to retrieve meta-information directly for an instance of some class `C` without having the constructor of `C` available, for example because the constructor is not accessible.

#### Example 7.3. Reflection with `N4Class`

This example demonstrates how these reflective features are accessed:

```
class A {}
class B extends A {}
var b = new B();
console.log(B.n4type.fqn);
console.log(b.constructor.n4type.fqn);
console.log(b.constructor.n4type.n4superType.fqn);
console.log(N4Type.of(b));
console.log(N4Type.of(B.n4type).fqn);
```

Assuming this code is defined in file `A`, this will output

```
A.B
A.B
A.A
A.B
N4BuiltInClasses.N4Class
```

The built-in types `N4Object` and `N4Class` are also accessible. They are not defined in a module, thus their *FQN* returns only their simple name.

#### Example 7.4. Reflection with Built-In Types

```
console.log('N4Object.n4class.fqn: ' + N4Object.n4class.fqn)
console.log('N4Class.n4class.fqn: ' + N4Class.n4class.fqn)

class A {}
console.log('A.n4class.fqn: ' + A.n4class.fqn)
console.log('A.n4class.n4superType.fqn: ' + A.n4class.n4superType.fqn)
```

Assuming this code is defined in file `A`, this will output

```
N4Object.n4class.fqn: N4Object
N4Class.n4class.fqn: N4Class
A.n4class.fqn: A.A
A.n4class.n4superType.fqn: N4Object
```

Note that classes extending `Object` do not provide the static `n4class` getter, that is

```
class B extends Object {}
console.log('B.n4class.fqn: ' + B.n4class.fqn)
```

would issue an error as cannot be resolved.

#### Example 7.5. `N4Class.of`

The type has a method to retrieve the meta-information from instances (i.e. or enumeration literals using ) without using the constructor.

```
class C { }
```

```

interface I {} class IImpl implements I {}
enum E { L }

var c: C = new C();
var i: I = new IImpl();
var e: E = E.L;

console.log(C.n4type.fqn);
console.log(N4Class.of(c).fqn);

console.log(I.n4type.fqn);
console.log(N4Class.of(i).fqn);

console.log(E.n4type.fqn);
console.log(N4EnumType.of(e).fqn);

```

## 7.4.2. Reflection for Interfaces

The meta-information of an interface `x` is available via getter `n4class` defined in the `type{x}`. This field is of type `N4Class` as well. Since an interface cannot have a super classs, the property `n4superTypes` will always be empty. Calling `isInterface` respectively on the returned `N4Class` instance will return true.

## 7.4.3. Reflection for Enumerations

```
var n: number; var b: boolean; var s: string;
```

The meta-information for enumerations is available by means of the getter `n4class`, either statically by using the enumeration type or (in terms of an instance getter) via a literal. Calling `isEnum` on the returned `N4Class` instance will return true.

## 7.5. Conversion of primitive types

Conversion between primitives is given as follows:

```
var n: number; var b: boolean; var s: string;
```

From	To	Conversion	Example
string	number	Number...	n = Number("42"); //42
string	boolean	N4Primitives.parseBoolean(...)	b=N4Primitives.parseBoolean("false");
number	boolean	Boolean(...)	b=Boolean(17.5); //true
number	string	Number.toString()	s=42.toString(); //"42"
boolean	number	N4Primitives.toNumber(...)	n=N4Primitives.toNumber(true);
boolean	string	Boolean.toString()	s=true.toString(); //"true" }

Remarks:

- ECMAScript doesn't define explicit conversion from string content. Implicit handling states all strings with `>0 == true`. `N4Primitives.parseBoolean(x)` yields true for `x.trim().toLowerCase().equals("true")`
- The call to `Boolean(..)` for the arguments `0`, `-0`, `null`, `false`, `NaN`, `undefined` and `""` evaluate to `false`. All other values evaluate to `true`.
- `Number` has several methods for converting a value to string [ECMA11a(p.S15.7.4)]: `toExponential()`, `toFixed()`, `toPrecision()`.
- ECMAScript doesn't define explicit conversion from boolean to number. Implicit handling states `true → 1` and `false → 0`, which `N4Primitives.toNumber()` yields.

# Chapter 8. Expressions

For all expressions, we define the following pseudo properties:

<code>containingExpression</code>	The parent expression, in which an expression is contained, may be null.
<code>containingStatement</code>	The statement in which the expression is (indirectly) contained.
<code>containingFunctionOrAccess</code>	The function, method, getter or setter in which the expression is (indirectly) contained, may be null.
<code>containingClass</code>	The class in which the expression is (indirectly) contained, may be null.
<code>probableThisTarget</code>	The potential target of a <code>this</code> keyword binding, this is not necessarily the containing class or object literal. In case of instance methods of a class <code>T</code> , this usually is the classifier <code>T</code> ; in case of static methods, it is the classifier type <code>type{type}</code> .
<code>container</code>	The direct owner of the expression. <code>z</code>

The expressions and statements are ordered, describing first the constructs available in the 5th edition of ECMA-262, referred to as [ECMA11a] in the following. It is worth noting that the grammar snippets already use newer constructs in some cases.

## 8.1. ECMAScript 5 Expressions

N4JS supports the same expressions as ECMAScript. The semantics are described in [ECMA11a(p.S11)]. In N4JS, some expressions are extended for supporting the declaration of types, annotations, or parameterized usages. These extensions and type-related aspects as well as specific N4JS constraints are described in this section.

Some operators come in different 'flavors', that is as binary operator, unary pre- or postfix operators, or assignment operators. For these operators, type constraints are only defined for the binary operator version and the other variants are deduced to that binary version. E.g., `++` and `+=` are deduced to `+` (and simple assignment).

### 8.1.1. The `this` Literal

This section describes the `this` literal and the semantics of the `@This` annotation, the type `this` is described in Section 4.12, "This Type".

#### Semantics

Semantics are similar to the original ECMAScript `this` keyword, see [ECMA11a(p.11.1.1, p.p.63)]). Also see [West06a] and MozillaJSRef

Regarding the location where `this` may be used, the following restrictions apply:

Valid location for this literal

**Requirement: IDE-173:** [Valid location for this literal](#) (ver. 1)

The literal may not be used in

1. the initializer expression of static data fields in classes.
2. the initializer expression of data fields in interfaces (applies to both static and non-static).
3. static methods of interfaces and static field accessors of interfaces.

See also [\[Req-IDE-69\]](#).

Note: [\[Req-IDE-173\]](#) also applies for this literals inside arrow expressions in initializers.

The use of `this` is illustrated with some examples as it can often be confusing. Type inference heuristics and explanations are provided in the next section.

#### Example 8.1. This in Unrestricted Mode

In unrestricted mode, `this` is bound to the receiver. If there is no receiver it is bound to the global object, however, we often do not know exactly what the global object would be.

```
var name = "global a"; // assume the top level is similar to the global object
this.name; // <-- "global a"
```

```

function f() {
    return this.name; // <-- depends on call, usually "global a"
}
var o11 = {
    name: "John",
    greeting: "Hello " + this.name, // "Hello global a" -- we do not greet John!
}
var o12 = {
    name: "John",
    f: function() {
        this.name; // usually "John", as we assume f is called like o12.f()
        var g = function() {
            return this.name; // "global a"
        }
        return g(); // no receiver, this in nested function g will be global scope
    }
}

```

**Example 8.2. This in strict mode**

In strict mode, `this` is bound to the receiver. If there is no receiver, it is bound to `undefined`. Thus, we will probably get a lot of errors:

```

"use strict"
var name = "global a"; // assume the top level is similar to the global object
this.name; // <-- error, this is undefined, there is no receiver
function f() {
    return this.name; // <-- depends on call, usually this produces an error as this is undefined
}
var o11 = {
    name: "John",
    greeting: "Hello " + this.name, // will produce an error, as this is undefined
}
var o12 = {
    name: "John",
    f: function() {
        this.name; // usually "John", as we assume f is called like o12.f()
        var g = function() {
            this.name; // an error, see call below:
        }
        return g(); // no receiver, this in nested function g is undefined
    }
}

```

**Example 8.3. This in N4JS mode**

As in strict mode, `this` is bound to the receiver and if there is no receiver, it is bound to `undefined`. So the example above is also true for N4JS mode. Classes behave slightly differently:

```

class A {
    name = "John";
    greeting = "Hello " + this.name; // works, in N4JS classes, greeting is "Hello John"

    f() {
        return this.name; // this usually is instance object, similar to object literals.
    }

    g() {
        var h = function() {
            return this.name; // as in object literals: no receiver, no this.
        }
        return h();
    }
}

```



In N4JS classes, `this` is always bound to the instance when used in field initialization.

**Type Inference**

The type is inferred from the `this` type is bound to. The inference, therefore, has to consider the original semantics as described in [ECMA11a(p.10.4., p.10.4.3, p.p.58)]. In ECMAScript the type of this is unfortunately determined by the function call and not by the function definition:

- By default, `this` is bound to the global object [ECMA11a(p.10.4.1.1)]. Unfortunately it is often unknown what the global object will be at run time (e.g., node.js differs from browsers).
- If a function is called without a receiver, `this` is bound to
  - the global object or
  - to `undefined` in strict mode.
- If a function is called with a receiver, `this` is bound to the receiver object.

Actually, `this` is bound to the newly created object if a function is called with the `new` operator. If a function is known to be invoked with an explicit `thisArg` (`apply()` etc.), the `@This` annotation can be used to explicitly set the this type. This annotation has precedence over otherwise inferred bindings.

#### Type Inference Heuristic for This-Keyword

##### Requirement: IDE-90: Type Inference Heuristic for This-Keyword (ver. 1)

In general, the actual this target can not be inferred from the context of the this keyword. A heuristic is defined, however, to compute the probable this type:

1. If the this keyword is used in some function annotated with an annotation `@This`, the type specified in the annotation is used. The inferred type is always nominal.

$$\begin{array}{l} f = "this".containingFunctionOrAccessor \\ \underline{f.hasAnnotation("@This") \quad T = f.annotation["@This"]} \\ \Gamma \vdash "this": \forall T \end{array}$$

2. If the this keyword is used in some `instance` method of a classifier or in an `instance` field initializer, `this` is bound to the `T` itself. If the this keyword is used in some `static` method of a classifier `T` or in a `static` field initializer, the prototype type (or constructor) of the classifier is used, that is `type[T]`. In both cases, the target is determined by using the expression's pseudo property `probableThisTarget`. If the this keyword is used in a function expression assigned to a property of an object literal, the type of the object literal is used. Note that usually this is the `this` type in instance methods, and the `this` type in static methods.

$$\begin{array}{l} T = "this".probableThisTarget \quad T \neq null \\ \underline{\Gamma \vdash "this": \forall T} \end{array}$$

3. In all other cases: Non-strict mode:

$$\begin{array}{l} mode = unrestricted \\ \underline{\Gamma \vdash "this": \text{global}} \end{array}$$

Strict mode and N4JS mode:

$$\begin{array}{l} mode \neq unrestricted \\ \underline{\Gamma \vdash "this": \text{global} \cup \text{undefined}} \end{array}$$

If the actual this type is defined as a structural type, the structural type information is moved to the this type itself. This is transparent to the user in general but maybe visible in case of error messages. That is to say that the actual this type is always a nominal type. This is indicated by the nominal modifier `\forall` (cf. [Req-IDE-90] constraints 1 and 2.).

#### Valid Target and Argument for @This Annotation

##### Requirement: IDE-91: Valid Target and Argument for @This Annotation (ver. 1)

1. The `@This` annotation is only allowed on declared functions, function expressions (including arrow functions), methods, and field accessors, i.e. getters and setters, except static members of interfaces.
2. The type declared by way of `@This(...)` an annotation of a method or field accessor must be a subtype of the member's containing classifier.

#### Single @This Annotation

##### Requirement: IDE-92: Single @This Annotation (ver. 1)

It is not allowed to use more than one `@This(...)` annotation on an element.

**Example 8.4. Effect of Nominal This Type**

Given the following declaration

```
@This(~Object with {a: string;}) f() {}
```

Since the this type is always nominal, `~ Object` becomes `Object`. In case of method call, however, the returned value becomes structural again. In case of error messages the type of the return type is then

```
~this[Object] with {a: string;}
```

For the sake of simplicity, additional structural members are usually omitted in error messages, leading to

```
~this[Object]
```

instead of

```
this[~Object]
```

**Example 8.5. This and Function Declaration**

This example demonstrates the usage of functions annotated with `@This`. By using the argument `union{A,B}` it is possible to have two completely unrelated classes as the receiver type of the function `logger`. To pass an actual object the `apply()` method of the function is used.

```
class A {
    log: string() { return "A was logged"; }
}

class B {
    log: string() { return "B was logged"; }
}

@This(union{A,B})
function logger() { console.log("~ "+this.log()+" ~"); }

var a: A = new A();
logger.apply(a,[]); // prints "~ A was logged ~"
logger.apply( new B(),[] ) // prints "~ B was logged ~"
```

**Example 8.6. This and Function Expressions**

In this example a function is created via a function expression. The function is then assigned to member field of class B. Via annotating the expression with `@This(B)`, access to the receiver of type B is enabled.

```
class B {
    log(): string { return "B was logged"; }      // method
    logMe : {@This(B) function():void}; // reference to a function
}

var b: B = new B();
b.logMe = @This(B) function() { console.log("*>"+this.log()+"<*"); }
b.logMe(); // prints *>B was logged<*
```

Note that if a function is called as a constructor function with `new`, the type of `this` can be declared via annotation `@This(...)`, as shown in the following snippet:

```
@This(
    ~Object with {
        w: number; h: number;
        area: {function():number};
    })
function Box(w: number w, h: number) {
    this.w = w;
    this.h = h;
    this.area = @This(
        ~Object with {
            w: number; h: number;
            area: {function():number};
        }) function() { return this.w * this.h }
}
```

```
var bError = Box(1,2)
var bOK = new Box(1,2)
```

Inside the constructor function `Box`, `this` is bound to the structural type definition due to the annotation.

Inside the nested function `area`, `this` is bound to the receiver object (if the function is called like `bOK.area()`). Again, this depends on the way the nested function is called, which can usually not be determined at the declaration location. The nested function must then be annotated accordingly.

When calling this function, the type of this is checked against the declared this type, which would cause an error in the first case.

The use of the `@This` annotation is not allowed on methods.



Using constructor functions is not recommended and an error or warning will be created. This is only useful for adapting third-party library code. Even in the latter case, it would probably make more sense to declare a (library) `class Rectangle` rather than defining the constructor function.

## 8.1.2. Identifier

### Syntax

Identifiers as expressions are identifier references. They are defined as follows:

```
IdentifierRef <Yield>:
  id=[types::IdentifiableElement|BindingIdentifier<Yield>]
;

BindingIdentifier <Yield>:
  IDENTIFIER
  | <!Yield> 'yield'
  | N4Keyword
;
```

### Semantics

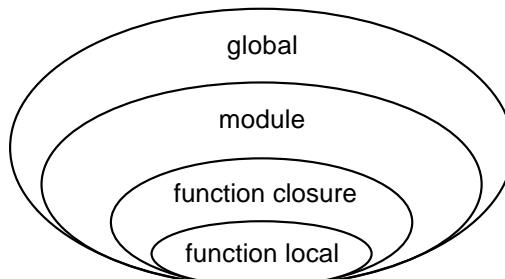
The type of an identifier *i* is resolved depending on its binding and scope respectively (cf.

[[ECMA11a\(p.10.2.2.1GetIdentifierReference, p.p.56\)](#)]). The following scopes (aka *Lexical Environments*) are defined:

- function local; local variables, parameters
- zero or more function closure in case of nested functions
- module
- global

These scope are nested as illustrated in [Figure 8.1, “Scopes”](#).

Note that classes definitions and object literal do not define a scope: members of a class or properties of an object literal are to be accessed via `this`. Identifier references always reference declared elements, that is to say either variable, function, or class declarations. Properties of object literals or members of a class are referenced via *PropertyAccess-Expression.property* (see [Section 8.1.7, “Property Accessors”](#)).



**Figure 8.1. Scopes**

An identifier may be bound to a variable (global or local variable, parameter, variable defined in a function's closure), or to a property of an object. The latter case is known as property access as further described in [Section 8.1.7, “Property Accessors”](#).

### Read Access to Identifier

**Requirement: IDE-93: Read Access to Identifier (ver. 1)**

If an identifier  $i$  is accessed, the bound declared element  $D$  must be readable if it is not used on the left-hand side of an assignment expression.

```
bind(i, D)
  ∧# AssignmentExpression ae ∈ i.container*:
    ae.left = i
    ∨ (μ(ae.left) = PropertyAccessExpression ∧ ae.left.property = i):
      ⇒ D.readable
```

### Type Inference

An identifier reference  $i$  is bound to an identifiable element  $i.id$ , which is expressed with the function  $bind(i, i.id)$ . The type of the reference is then inferred as follows:

$$\frac{\Gamma \vdash idref.id:T}{\Gamma \vdash \text{IdentifierRef } idref:T}$$

### 8.1.3. Literals

cf. [ECMA11a(p.S11.1.3p.63, p.S7.8p.19ff)].

#### Type Inference

The type of a literal can directly be derived from the grammar. The following axioms are defined for literals:

NullLiteral:null  
BooleanLiteral:boolean  
NumericLiteral:intornumber  
StringLiteral:string  
RegularExpressionLiteral:RegExpr

Note that there are no literals specific for `pathSelector` or `i18nkey`.

#### 8.1.3.1. Integer Literals

Numeric literals representing integers in the range of JavaScript's int32 are inferred to the built-in primitive type `int` instead of `number`. The following rules apply:

##### Numeric literals

**Requirement: IDE-94: Numeric literals (ver. 1)**

- Numeric literals with a fraction or using scientific notation, e.g. `2.0` and `2e0`, respectively, are always inferred to `number`, even if they represent integers in the range of int32.
- Numeric literals that represent integers in the range of JavaScript's int32, i.e. from  $-2^{31}$  to  $2^{31}-1$ , are inferred to `int`.
- Hexadecimal and octal literals are always interpreted as positive numbers, so all values above `0x7fffffff` and `017777777777` lie outside the range of int32 and will thus be inferred to `number`; this is an important difference to Java. See below for further elaboration.

There are differences to numeric literals in Java:

Literal	Java		JavaScript N4JS	
	Value	Type	Value	Type
<code>2147483648</code>	<code>-2147483648</code>	<code>int</code>	<code>-2147483648</code>	<code>int</code>
<code>2147483647</code>	<code>2147483647</code>	<code>int</code>	<code>2147483647</code>	<code>int</code>
<code>0x7fffffff</code>	<code>2147483647</code>	<code>int</code>	<code>2147483647</code>	<code>int</code>

	Java		JavaScript N4JS	
0x800000000	-2147483648	int	+2147483648	number
0xffffffff	-1	int	4294967295	number
0x100000000	n/a		4294967296	number
017777777777	2147483647	int	2147483647	int
0200000000000	-2147483648	int	+2147483648	number
0377777777777	-1	int	4294967295	number
0400000000000	0	int	4294967296	number
01000000000000	n/a		8589934592	number

The literals `0x100000000` and `01000000000000` produce a syntax error in Java.

Until IDE-1881 is complete, all built-in operations always return a `number` even if all operands are of type `int`. For the time being, we therefore interpret `-1` as a negative integer literal (inferred to `int`), but `-(1)` as the negation of a positive integer literal (inferred to `number`).

## 8.1.4. Array Literal

### Syntax

cf [ECMA11a(p.S11.1.4, p.p.63)]

```
ArrayLiteral <Yield> returns ArrayLiteral:
  {ArrayLiteral} '['
    elements+=ArrayPadding* (
      elements+=ArrayElement<Yield>
      (',' elements+=ArrayPadding* elements+=ArrayElement<Yield>)*
      (trailingComma? ',' elements+=ArrayPadding*)?
    )?
  ']'
;

/**
 * This array element is used to pad the remaining elements, e.g. to get the
 * length and index right
 */
ArrayPadding returns ArrayElement: {ArrayPadding} ',';

ArrayElement <Yield> returns ArrayElement: {ArrayElement} spread?= '...'?
  expression=AssignmentExpression<In=true,Yield>;
```

### Type Inference

In general, an array literal is inferred as `Array<T>` (similar to the type of `new Array()`). The interesting question is the binding of the type variable `T`.

The type of an array padding `p` is inferred as follows:

$$\Gamma \vdash p : \text{undefined}$$

The element type of an array literal is simply inferred as the (simplified) union of the type elements of the array. Thus, the type of an array literal `a` is inferred as follows:

$$\frac{\Gamma \vdash a : \text{elements}: T_e \quad T = \bigcup T_e}{\Gamma \vdash (a) : \text{Array}<T>}$$

In other languages not supporting union types, the element type is often inferred as the join (LCST) of the element types. Using a union type here preserves more information (as the actual types are still known). For many use cases the behavior is similar though, as the members of a union type are the members of the join of the elements of the union.

Note that `typeof [1,2,3]` does not return `Array<number>` (as ECMAScript is not aware of the generic array type), but `object`.

#### Example 8.7. Array Type Inference

The type for all variables declared in this example is inferred to `Array<string>`:

```
var names1      = ["Walter", "Werner"];
var names2      = new Array("Wim", "Wendelin");
var names3      = new Array<string>(3); // length is 3
var names4: Array<string>;
```

Empty array literals are inferred to `any`, by default. We are not using `Array<?>` here because then a typical JavaScript pattern would no longer be supported:

```
var a = [];
a.push('hello'); ①
```

① This would fail if `a` and thus `[]` were inferred to `Array<?>`



An important exception: if a type expectation exists for the empty array literal and the expected type is `Array<T>`, this will be used as the type of the array literal.

### Empty array literal

#### Requirement: IDE-95: Empty array literal (ver. 1)

An empty array literal will be inferred as follows:

- If there is a type expectation for the empty array literal and the expected type is `Array<T>`, for any type `T`, then the type of the empty array literal will be inferred to `Array<T>`.
- Otherwise, the type of the empty array literal will be inferred to `Array<any>`.

## 8.1.5. Object Literal

### Syntax

Cf. [ECMA11a(p.S11.1.5, p.p.65ff)] The syntax of an object literal is given by:

```
ObjectLiteral <Yield>: {ObjectLiteral}
  '{'
    ( propertyAssignments+=PropertyAssignment<Yield>
      (',', propertyAssignments+=PropertyAssignment<Yield>)* ','?
    )?
  '}'
;

PropertyAssignment <Yield>:
  PropertyNameValuePair<Yield>
  | PropertyGetterDeclaration<Yield>
  | PropertySetterDeclaration<Yield>
  | PropertyMethodDeclaration<Yield>
  | PropertyNameValuePairSingleName<Yield>
;

PropertyMethodDeclaration <Yield>:
  => ({PropertyMethodDeclaration})
    annotations+=Annotation*
    TypeVariables? returnTypeRef=TypeRef?
    (
      generator?=/* LiteralOrComputedPropertyName<Yield> ->MethodParamsAndBody<Generator=true>
      | LiteralOrComputedPropertyName<Yield> ->MethodParamsAndBody <Generator=false>
    )
  )
  ?/??
;

PropertyNameValuePair <Yield>:
  => (
    {PropertyNameValuePair}
    annotations+=Annotation*
    declaredTypeRef=TypeRef? LiteralOrComputedPropertyName<Yield> ':'
  )
  expression=AssignmentExpression<In=true,Yield>
;

/*
```

```

* Support for single name syntax in ObjectLiteral (but disallowed in actual object literals by ASTStructureValidator)
* except in assignment destructuring patterns)
*/
PropertyNameValuePairSingleName <Yield>:
    declaredTypeRef=TypeRef?
    identifierRef=IdentifierRef<Yield>
    ('=' expression=AssignmentExpression<In=true,Yield>)?  

;  

PropertyGetterDeclaration <Yield>:
    =>(
        {PropertyGetterDeclaration}
        annotations+=Annotation*
        GetterHeader<Yield>
    )
    body=Block<Yield=false>  

;  

PropertySetterDeclaration <Yield>:
    =>(
        {PropertySetterDeclaration}
        annotations+=Annotation*
        'set'
        ->LiteralOrComputedPropertyName <Yield>
    )
    ('(' fpar=FormalParameter<Yield> ')') body=Block<Yield=false>  

;  

import Address from "my/Address";
var simple = {name: "Walter", age: 72, address: new Address()};

```

### 8.1.5.1. Properties

PropertyAssignments have common properties of PropertyNameValuePair, PropertyGetterDeclaration, and PropertySetterDeclaration:

**annotations** The annotations of the property assignment.

**name** The name of the property. This may be an identifier, a string or a numeric literal. When comparing names, we implicitly assume the name to be converted to an identifier, even if this identifier is not a valid ECMAScript identifier.

**declaredType** The declared type of the property which may be null. This property is a pseudo property for PropertySetterDeclaration, in this case it is derived from the declared type of the setter's formal parameter.

Additionally, we introduce the following pseudo properties to simplify constraints:

**isAccessor** The read-only boolean property. This is true if the property assignment is a setter or getter declaration. This is comparable to ECMAScript's spec function `IsAccessoprDescriptor`. For a given property assignment  $p$  this is semantically equivalent to  $\mu(p) = \text{PropertyGetterDeclaration} \vee \mu(p) = \text{PropertySetterDeclaration}$ .

**isData** The read-only boolean property. This is true if the property assignment is a name value pair. For a given property assignment  $p$  this is semantically equivalent to  $\mu(p) = \text{PropertyNameValuePair}$ . It is comparable to ECMAScript's spec function `isDataDescriptor`. The equation  $\text{isAccessor} = \neg \text{isData}$  is always true.

### Semantics

#### Object literal

##### Requirement: IDE-96: Object literal (ver. 1)

For a given object literal  $ol$  the following constraints must hold (cf. [ECMA11a(p.p.66)]):

- the name of each property is given as an identifier, a string literal, a numeric literal, or as a computed property name with a compile-time expression (see Section 8.5, “Compile-Time Expressions”). In particular, string literals, e.g. `[ 'myProp' ]`, built-in symbols, e.g. `[Symbol.iterator]`, and literals of `@StringBased` enums are all valid computed property names.
- Object literal may not have two PropertyNameValuePair with the same name in strict mode (cf. 4.a [ECMA11a(p.p.66)]):

$mode = \text{strict} \rightarrow \forall pa \in ol.propertyAssignments, paisData:$   
 $\# pa \in ol.propertyAssignments:$   
 $pa.isAccessor \wedge pa.name = pa.name$

- Object literal may not have `PropertyNameValuePair` and `PropertyGetterDeclaration/PropertySetterDeclaration` with the same name (cf. 4.b/c [[ECMA11a\(p.p.66\)](#)]):
 
$$\forall pa \in ol.propertyAssignments, pa \neq \text{isData} : \\ \nexists pg \in ol.propertyAssignments : \\ \mu(pg) \neq \text{PropertyNameValuePair} \wedge pg.name = p.name$$
- Object literal may not have multiple `PropertyGetterDeclaration` or `PropertySetterDeclaration` with the same name (cf. 4.d [[ECMA11a\(p.p.66\)](#)]):
 
$$\forall pg \in ol.propertyAssignments, pg \neq \text{isAccessor} : \\ \nexists pg' \in ol.propertyAssignments \{ pg \} : \\ \mu(pg') = \mu(pg) \wedge pg'.name = pg.name$$

It is a `SyntaxError` if the Identifier `eval` or the Identifier `arguments` occurs as the Identifier in a `PropertySetParameterList` of a `PropertyAssignment` that is contained in strict code or if its `FunctionBody` is strict code. [[ECMA11a\(p.p.66\)](#)]

- If two or more property assignments have the same name (and the previous conditions hold), then the types of these assignments must *conform*. That is to say that the inferred (but not declared) type of all assignments must be type of probably declared types and if the types are explicitly declared, they must be equal.
- In N4JS mode, the name of a property must be a valid N4JSIdentifier:

`mode=n4js → ∀ pa ∈ ol.propertyAssignments :`  
 $\mu(p.name) = \text{N4JSIdentifier}$

#### Superfluous properties of an object literal

**Requirement: IDE-22501: Superfluous properties of an object literal (ver. 1)**

Let  $E$  be the expected type of an object literal  $O$  as defined by the context in which  $O$  is used. If  $E$  is not type `Object` and not dynamic, then the compiler creates a warning  $O$  contains properties not found in  $E$ .

This is true in particular for object literals passed in as arguments of a spec-constructor.

#### 8.1.5.2. Scoping and linking

##### Example 8.8. Scoping and linking

```
var p = {
  f: function() {
    console.log("p's f");
  },
  b: function() {
    this.f();
  },
  o: {
    nested: "Hello"
  }
};
p.b();
p.o.nested;
```

- Other properties within an object literal property can be accessed using `this`. In the expression of property name value pairs, however, `this` is not bound to the containing object literal, but usually to `undefined` or `global`.
- The properties of an object literal are accessible from outside.
- Nested properties of an object literal are also accessible from outside.

#### Type Inference

An object literal implicitly extends `~Object`, therefore, object literal types use structural typing. For details see [Section 5.3, "Structural Typing"](#). From a type systems point of view, the two variables `ol` and `st` below have the same type.

```
var ol = {
  s: "hello",
  n: 42
}
var st: ~Object with { s: string; n: number; };
```

## 8.1.6. Parenthesized Expression and Grouping Operator

The grouping operator is defined here as a parenthesized expression.

### Syntax

cf. [ECMA11a(p.S11.1.6, p.p.67)]

```
ParenExpression <Yield>: '(' expression=Expression<In=true,Yield> ')';
```

### Type Inference

The type of the grouping operator simply is the type of its nested expression. The type if a parenthesized expression *pe* is inferred as follows:

$$\frac{\Gamma \vdash e : T}{\Gamma \vdash ('e') : T}$$

#### Example 8.9. Parenthesized Expression Type Examples

In the following listing, the type of the plain expressions is equivalent to the parenthesized versions:

```
class A{} class B extends A{}
var f: boolean; var a: A a; var b: B;

/* simple      <->      parenthesized */
10;          (10);
"hello";     ("hello");
true;         (true);
a;           (a);
10-5;        (10-5);
f?a:b       (f?a:b);
```

## 8.1.7. Property Accessors

### Syntax

Property accessors in N4JS are based on [ECMA11a(p.S11.2.1, p.p.67ff)]. They cannot only be used for accessing properties of an object, but also for accessing members of a class instance. In order to support parameterized calls, the syntax is extended to optionally allow type arguments.

```
ParameterizedPropertyAccessExpression:
    target=PrimaryExpression<Yield> ParameterizedPropertyAccessExpressionTail<Yield>
;

IndexedAccessExpression:
    target=PrimaryExpression<Yield> IndexedAccessExpressionTail<Yield>
;

fragment IndexedAccessExpressionTail <Yield>*:
    '[' index=Expression<In=true,Yield> ']'
;

fragment ParameterizedPropertyAccessExpressionTail <Yield>*:
    '.' TypeArguments? property=[types::IdentifiableElement / IdentifierName]
;
```

Note that in [ECMA11a], the `index access` is called `bracket notation`.

### Direct Property Access

We define a special case of property access as follows:

#### Direct Property Access

##### Definition: Direct Property Access

A property access expression is called *direct*, iff

- its target is an identifier reference to a class, interface, enum, or the built-in object `Symbol`, and

- its property name denotes an owned member of the target classifier (not an inherited, consumed, or polyfilled member) or a literal if the target is an enum.

As a consequence, a direct property access can only refer to static members.

The first requirement of the above definition rules out property access expressions that do not directly point to their target classifier or enum, as shown in the following example:

```
class C {
  const field = 'hello';
}
C.field; // direct property access to 'field'
let ctor = C;
ctor.field; // *not* a direct property access to 'field'
```

Direct property access is the only form of property access allowed in compile-time expressions, cf. [Section 8.5, “Compile-Time Expressions”](#).

### 8.1.7.1. Properties

We define the following properties:

- target** The receiver of the property access.
- index** The index expression in case of an `IndexedAccessExpression` (returns `null` otherwise).
- property** The name of the property in case of non-indexed-access expressions (returns `null` otherwise, although the index may be interpreted as property name).

We define the following pseudo properties:

- isDotAccess** Read-only boolean property, returns true for non-index access expression (similar to  $\mu(p) \neq \text{IndexedAccessExpression}$ ).
- isIndexAccess** Read-only boolean property, returns true for index access expression (similar to  $\mu(p) = \text{IndexedAccessExpression}$ ).  
The equation  $p.\text{isDotAccess} = \neg p.\text{isIndexAccess}$  is always true.
- name** Returns the name of the property. This is either the *property* converted to a simple name or the index converted to a name (where possible) if it is an indexed-accessed expression.

## Semantics

The parameterization is part of the property access in case of generic methods. For generic functions, a parameterized function call is introduced (cf. [Section 8.1.10, “Function Calls”](#)). The constraints are basically similar.

### Property Access and Dot Notation

#### Requirement: IDE-97: Property Access and Dot Notation (ver. 1)

- If dot notation is used in N4JS mode, the referenced property must exist unless receiver is a dynamic type:

$$pae.\text{isDotAccess} \wedge \neg R.\text{dyn} \rightarrow \exists m \in pae.\text{target.type.properties}: m.\text{name} = pae.\text{name}$$

- If dot notation is used and the referenced property exists, then the property must be accessible:

$$pae.\text{isDotAccess} \wedge \neg R.\text{dyn} \rightarrow (\exists m \in pae.\text{target.type.properties}: m.\text{name} = pae.\text{name}) \rightarrow \alpha(pae, m)$$

- If dot notation is used and the referenced property exists and this property is a member with a declared `@This` type (only possible for methods or field accessors), then the receiver must be a subtype of the declared `@This` type.

### Index Access

#### Requirement: IDE-98: Index Access (ver. 1)

An index access expression is valid iff one of the following cases applies:

- the receiver is of a dynamic type. In this case, the index may be any expression (need not be a compile-time expression).
- the receiver is an immediate instance of `Object`, i.e. it is a subtype of `Object` and its super types but **not** of any other type including `~Object` and `~~Object`.

3. the receiver is of type Array, ArgumentType, string, or String (including their subtypes) **and** the index is an expression of type `number`.
4. the index expression is a compile-time expression
  - **and** the receiver type defines a member with a name equal to the string representation of the index expression's compile-time value
  - **and** the receiver is not an enum.

Although index access is very limited, it is still possible to use immediate instances of `Object` in terms of a map (but this applies only to index access, not the dot notation):

#### Example 8.10. Object as Map

```
var map: Object = new Object();
map[ "Kant" ] = "Imperative";
map[ "Hegel" ] = "Dialectic";
map.spinoza = "Am I?"; // error: Couldn't resolve reference to IdentifiableElement 'spinoza'.
```

#### Parameterized Property Access

##### Requirement: IDE-99: Parameterized Property Access (ver. 1)

For a parameterized property access expression *pae*, the following constraints must hold:

1. The receiver or target must be a function or method:  
 $pae.target.type < : Function$
2. The number of type arguments must match the number of type parameters of the generic function or method:  
 $|pae.typeArgs| = |pae.target.typeVars|$
3. The type arguments of a parameterized property access expression must be subtypes of the boundaries of the parameters of the called generic method.

Also see constraints on read ([Req-IDE-93]) and write ([Req-IDE-121]) access.

#### Type Inference

Cf. [ECMA11a(p.S11.2.1, p.p.67ff)]

We define the following type inferencing rules for property accessors:

- The type of an indexed-access expression *p* is inferred as follows:

$$\frac{\neg p.target.dynvp.index.type < : [number] \quad \Gamma \vdash p.target: \text{Array}\langle T \rangle}{\frac{\Gamma \vdash p:T}{\frac{\text{else}}{\Gamma \vdash p:\text{any}}}}$$

- The type of a property access expression is inferred as follows:

$$\frac{\Gamma \leftarrow \theta(R) \vdash expr.target: R \quad \Gamma \vdash expr.property: T}{\text{PropertyAccessExpression } expr: T}$$

- The type of a parameterized access expression *p* is inferred as follows:

$$\frac{\exists m \in p.target: m.name = p.name \quad \Gamma \vdash m: T}{\frac{\Gamma \vdash p:T}{\Gamma \vdash p:\text{any}}}$$

#### 8.1.8. New Expression

Cf. [ECMA11a(p.S11.2.2, p.p.68)]

#### Syntax

```
NewExpression: 'new' callee=MemberExpression<Yield> ( -> TypeArguments )?
  ( => withArgs?= '(' Arguments<Yield>? ')' )?
```

```
import Address from "my/Address";
```

```

var a = new Address();
// a.type := my/Address

class C<T> {
    constructor(param: T) {}
}

var c = new C<string>("hello");

```

## Semantics

### New expression

#### Requirement: IDE-100: New expression (ver. 1)

Let  $ne$  be a new expression, with  $\Gamma \vdash ne.callee:C$ . The following constraints must hold:

1. The callee must be a constructor type:  $C < :constructor\{?\}$  or a constructable type.
2. Let  $O$  be the type argument of  $C$ , that is  $C = constructor\{O\}$ . In that case,
  - a.  $O$  must not be an interface or enum:  $\mu(C) \notin \{\text{Interface}, \text{Enum}\}$
  - b.  $O$  must not contain any wildcards.
  - c.  $O$  must not be a type variable.
3. If  $C$  is not a constructor type, it must be a constructable type, that is one of the following:

```
{Object, Function, String, Boolean,}
{Number, Array, Date, RegExp, Error}
```

In particular, it must not refer to a primitive type or a defined functions (i.e., subtypes of `Function`) cannot be used in new-expressions in N4JS.

### Remarks:

to 1 The type of an abstract class `A` is  $type\{A\}$ . Or in other words: Only instantiable classes have an inferred type of  $constructor\{\dots\}$ .

to 2 Even though it is possible to use the constructor type of an abstract class – concrete subclasses with override compatible constructor signature will be subclasses of this constructor.

to 3 It is not possible to refer to union or intersection at that location. So this is not explicitly denied here since it is not possible anyway.

#### Example 8.11. Abstract classes and construction

The following examples demonstrates the usage of abstract classes and constructor types, to make the first two constraints more clearer:

```

/* XPECT_SETUP org.eclipse.n4js.spec.tests.N4JSSpecTest END_SETUP */

abstract class A {}

class B extends A {}

// XPECT errors --> "Cannot instantiate abstract class A." at "A"
var x = new A();
// XPECT noerrors -->
var y = new B();

function foo(ctor : constructor{A}) {
    // XPECT noerrors -->
    return new ctor();
}

// XPECT errors --> "type{A} is not a subtype of constructor{A}." at "A"
foo(A);
// XPECT noerrors -->
foo(B);

```

## Type Inference

The type of a new expression  $ne$  is inferred as follows:

$$\frac{\Gamma \vdash ne.\text{callee}:\text{constructorC}}{\Gamma \vdash ne:C}$$

For classes, constructors are described in [Section 4.11, “Constructor and Classifier Type”](#).

In N4JS it is not allowed to call new on a plain function. For example:

```
function foo() {}
var x = new foo();
```

will issue an error.

## 8.1.9. Function Expression

See [Chapter 6, Functions](#) for details.

## 8.1.10. Function Calls

In N4JS, a function call [[ECMA11a\(p.S11.2.3\)](#)] is similar to a method call. Additionally to the ECMAScript's CallExpression, a ParameterizedCallExpression is introduced to allow type arguments passed to plain functions.

### Syntax

```
[[function-calls-syntax]]
```

Similar to [[ECMA11a\(p.S11.2.3, p.p.68ff\)](#)], a function call is defined as follows:

```
CallExpression <Yield>:
  target=IdentifierRef<Yield>
  ArgumentsWithParentheses<Yield>
;

ParameterizedCallExpression <Yield>:
  TypeArguments
  target=IdentifierRef<Yield>
  ArgumentsWithParentheses<Yield>
;

fragment ArgumentsWithParentheses <Yield>*:
  '(' Arguments<Yield>? ')'
;

fragment Arguments <Yield>*:
  arguments+=AssignmentExpression<In=true,Yield> (',' arguments+=AssignmentExpression<In=true,Yield>)* (',' spread?=...)
  arguments+=AssignmentExpression<In=true,Yield>?
    | spread?=... arguments+=AssignmentExpression<In=true,Yield>
;
```

### Semantics

#### Function Call Constraints

##### **Requirement: IDE-101: Function Call Constraints (ver. 1)**

For a given call expression  $f$  bound to a method or function declaration  $F$ , the following constraints must hold:

- If less arguments are provided than formal parameters were declared, the missing formal parameters must have been declared optional:  
 $|f.args| < |F.pars| \rightarrow \forall |f.args| < i \leq |F.pars|: F.pars_i: \text{optional}$
- If more arguments are provided than formal parameters were declared, the last formal parameter must have been declared variadic:  
 $|f.args| > |F.pars| \rightarrow F.pars_{|F.pars|+1}: \text{variadic}$
- Types of provided arguments must be subtypes of the formal parameter types:  
 $\forall 0 < i < \min(|f.args|, |F.pars|): f.args_i < :F.pars_i$
- If more arguments are provided than formal parameters were declared, the type of the exceeding arguments must be a subtype of the last (variadic) formal parameter type:  
 $\forall |F.pars| < i \leq |f.args|: f.args_i < :F.pars_{|F.pars|+1}$

## Parameterized Function Call Constraints

### Requirement: IDE-102: Parameterized Function Call Constraints (ver. 1)

\* The number of type arguments in a parameterized call expression must be equal to the number of type parameters of the generic function / method and the type arguments must be subtypes of the corresponding declared upper boundaries of the type parameters of the called generic function.

Note that (for a limited time), constraints [Req-IDE-101] and [Req-IDE-102] are not applied if the the type of  $F$  is `Function`. See Section 4.7.3, “Function-Object-Type”.

## Type Inference

A call expression  $expr$  is bound to a method (Section 5.2.3, “Methods”) or function declaration (which may be part of a function definition (Section 6.2.1, “Function Declaration” or specified via a function type Section 6.1, “Function Type”)  $F$  (via evaluation of `MemberExpression`). The type of the call is inferred from the function declaration or type  $F$  as follows:

$$\frac{\text{bind}(\text{expr}.\text{target}, F) \quad F.\text{returnType}:T}{\Gamma \vdash \text{expr}:T}$$

A generic method invocation may be parameterized as well. This is rarely required as the function argument types are usually inferred from the given arguments. In some cases, for instance with pathSelectors, this is useful. In that case, the type variable defined in the generic method declaration is explicitly bound to types by using type arguments. See Section 8.1.7, “Property Accessors” for semantics and type inference.

### Example 8.12. Generic Method Invocation

This examples demonstrate how to explicitly define the type argument in a method call in case it cannot be inferred automatically.

```
class C {
    static <T> foo(p: pathSelector<T>): void {...}
};

C.<my.Address>foo("street.number");
```

Note that in many cases, the type inferencer should be able to infer the type automatically. For example, for a method

```
function <T> bar(c: T, p: pathSelector<T>): void {...};
```

and a function call

```
bar(context, "some.path.selector");
[source,n4js]
```

the type variable `T` can be automatically bound to the type of variable `context`.

## 8.1.11. Postfix Expression

### Syntax

```
PostfixExpression returns Expression: LeftHandSideExpression
    (=>({PostfixExpression.expression=current} /* no line terminator here */ op=PostfixOperator))?
;
enum PostfixOperator: inc='++' | dec='--';
```

### Semantics and Type Inference

The type inference and constraints for postfix operators `++` and `--`, cf. [ECMA11a(p.S11.3.1, p.p.70)], [ECMA11a(p.S11.3.1, p.p.70)], are defined similarly to their prefix variants (unary expressions), see Section 8.1.12, “Unary Expression”.

## Postfix Expression Constraints

### Requirement: IDE-103: Postfix Expression Constraints (ver. 1)

For a given postfix expression  $u$  with  $uop \in \{++, --\}$  and  $uexpression:type:T$ , the following constraints must hold:

- In N4JS mode, the type  $T$  of the expression must be a number.

- If  $uexpression = \text{PropertyAccess } pa(p) \wedge p \text{ is Dot Access} \rightarrow$  both *get p* and *set p* must be defined.

## 8.1.12. Unary Expression

### Syntax

We define the following unary operators and expression, similar to [ECMA11a(p.p.70ff)]:

```
UnaryExpression returns Expression:
  PostfixExpression
  | ({UnaryExpression} op=UnaryOperator expression=UnaryExpression);
enum UnaryOperator: delete | void | typeof | inc='++' | dec='--' | pos='+' | neg='-' | inv='$\\sim$' | not='!';

```

### Semantics

For semantics of the *delete* operator, see also [MozillaJSRef]

#### Delete Operator Constraints

##### Requirement: IDE-104: Delete Operator Constraints (ver. 1)

For a given unary expression  $u$  with  $uop = \text{delete}$ , the following constraints must hold:

- In strict mode,  $uexpression$  must be a reference to a property of an object literal, a member of a class type, or to a property of the global type (i.e., the reference must be bound, and the bound target must not be a variable).
- In N4JS mode, the referenced property or member must not be declared in the containing type and the containing type reference must be declared dynamic.

#### Void Operator Constraints

##### Requirement: IDE-105: Void Operator Constraints (ver. 1)

There are no specific constraints defined for with  $uop = \text{void}$

#### Typeof Operator Constraints

##### Requirement: IDE-106: Typeof Operator Constraints (ver. 1)

There are no specific constraints defined for unary expression  $u$  with  $uop = \text{typeof}$ .

#### Increment/Decrement Constraints

##### Requirement: IDE-107: Increment/Decrement Constraints (ver. 1)

For a given unary expression  $u$  with  $uop \in \{++, --\}$  and  $uexpression.type:T$ , the following constraints must hold:

- If mode is N4JS, the type  $T$  of the expression must be a number

$\Gamma \vdash \text{UnaryExpression} \triangleleft \text{Expression: number}$

- If  $uexpression = \text{PropertyAccess } pa(p) \wedge p \text{ is Dot Access} \rightarrow$  both *get p* and *set p* must be defined.

#### Unary Plus/MINUS/Bitwise Not Operator Constraints

##### Requirement: IDE-108: Unary Plus/MINUS/Bitwise Not Operator Constraints (ver. 1)

For a given unary expression  $u$  with  $uop \in \{+, -, \sim\}$  and  $uexpression.type:T$ , the following constraints must hold:

- In N4JS mode, the type  $T$  of the expression must be a number:

$\Gamma \vdash \text{UnaryExpression} \triangleleft \text{Expression: number}$

#### Logical Not Operator Constraints

##### Requirement: IDE-109: Logical Not Operator Constraints (ver. 1)

There are no specific constraints defined for with  $uop = !$

## Type Inference

The following operators have fixed types independent of their operand types:

$$\begin{aligned}\Gamma \vdash \text{'delete'} \ expression: \text{boolean} \\ \Gamma \vdash \text{'void'} \ expression: \text{undefined} \\ \Gamma \vdash \text{'typeof'} \ expression: \text{string} \\ \Gamma \vdash ('+' '-' '-' '+' '-' '-' ') \ expression: \text{number} \\ \Gamma \vdash '!' \ expression: \text{boolean}\end{aligned}$$

### 8.1.13. Multiplicative Expression

#### Syntax

Cf. [ECMA11a(p.p.73ff)]

```
MultiplicativeExpression returns Expression: UnaryExpression
  (=>({MultiplicativeExpression.lhs=current} op=MultiplicativeOperator) rhs=UnaryExpression)*;
enum MultiplicativeOperator: times='*' | div='/' | mod='%' ;
```

#### Semantics

##### Multiplicative Expression Constraints

**Requirement: IDE-110: Multiplicative Expression Constraints** (ver. 1)

For a given multiplicative expression the following constraints must hold in N4JS mode :

- The types of the operands may be any type:

$$\Gamma \vdash \text{MultiplicativeExpression} \triangleleft \text{Expression: any}$$

If a non-numeric operand is used, the result may be `Nan` which actually is a number as well.

#### Type Inference

```
[[type-inference-9]]
```

The inferred type of a multiplicative expression always is number:

$$\Gamma \vdash \text{MultiplicativeExpression: number}$$

### 8.1.14. Additive Expression

#### Syntax

Cf. [ECMA11a(p.p.75ff)]

```
AdditiveExpression returns Expression: MultiplicativeExpression
  (=>({AdditiveExpression.lhs=current} op=AdditiveOperator) rhs=MultiplicativeExpression)*;
enum AdditiveOperator: add='+' | sub='-' ;
```

#### Semantics

##### Additive Expression Constraints

**Requirement: IDE-111: Additive Expression Constraints** (ver. 1)

For a given additive expression the following constraints must hold in N4JS mode:

- The type of the operand can be any type:

$$\Gamma \vdash \text{AdditiveExpression} \triangleleft \text{Expression: any}$$

In JavaScript it is possible to subtract two non-numerics, leading to `Nan`. Also `undefined` or `null` may be used. The real difference is what type is to be returned (string or number, see below).

### 8.1.14.1. Type Inference

The type of an additive expression is usually inferred to `number`, except for addition which may lead to string as well. The result for the addition operator is only be a number if both operands are numbers, booleans, null, or undefined. Using `undefined` in an additive expression leads to `NaN` which actually is a number from the type system's point of view. Additional analysis may create errors in the latter case though.

We first define two helper rules to simplify the addition operator condition:

$$\begin{array}{c} \exists N \in \{\text{number}, \text{int}, \text{boolean}, \text{null}, \text{undefined}\}: T <: N \\ \text{nb}(T) \end{array}$$

$$\frac{\text{nb}(T) \vee (\mu(T) = \text{Union} \wedge \exists E \in T.\text{typeRefs}: mnb(E))}{mnb(T)} \quad \text{mnb}$$

$$\frac{\Gamma \vdash e.\text{lhs}: L \quad \Gamma \vdash e.\text{rhs}: R \quad \text{nb}(L) \quad \text{nb}(R)}{\text{toNum(expr)}} \quad \text{toNum}$$

$$\frac{\Gamma \vdash e.\text{lhs}: L \quad \Gamma \vdash e.\text{rhs}: R \quad \text{mnb}(L) \quad \text{mnb}(R)}{\text{mayNum(expr)}} \quad \text{mayNum}$$

The type of an additive expression  $e$  is inferred as follows:

$$\frac{e.\text{op} = '+' \quad \neg \text{toNum}(e) \quad \neg \text{mayNum}(e)}{\Gamma \vdash e: \text{string}}$$

$$\frac{e.\text{op} = '+' \quad \neg \text{toNum}(e) \quad \text{mayNum}(e)}{\Gamma \vdash e: \text{union}\{\text{number}, \text{string}\}}$$

$$\frac{e.\text{op} = '+' \quad \text{toNum}(e)}{\Gamma \vdash e: \text{number}}$$

$$\frac{e.\text{op} \leq '+'}{\Gamma \vdash e: \text{number}}$$

That is, if both operands are number, int, boolean, null, or even undefined, then the 'plus' is interpreted as mathematical addition and the result is a number. In other cases the 'plus' is interpreted as string concatenation and the result is a string. In case of union types, the result may be a union of number and string.

Adding two integers (int) leads to a number, since the result may not be represented as an (JavaScript) int anymore.

#### Example 8.13. Type of addition expression

```
1+2;           // number 3
"1"+"2";      // string "12"
"1"+2;         // string "12"
1+true;        // number 2
false+1;       // number 1
"1"+true;      // string "ltrue"
"1"+null;      // string "lnull"
1+null;        // number 1
1+undefined;   // number NaN
"1"+undefined; // string "lundefined"
```

Support new `Symbol.toPrimitive`.

### 8.1.15. Bitwise Shift Expression

#### Syntax

```
Cf. +[ +<<ECMA11a,ECMA11a(p.p.76f)>>+]+
```

```
ShiftExpression returns Expression: AdditiveExpression
  (=>({ShiftExpression.lhs=current} op=ShiftOperator rhs=AdditiveExpression))*;
;

ShiftOperator returns ShiftOperator:
  '><' '><' '>?' // SHR, USHR
  | '<>' '<>' // SHL
;
```

#### Semantics

##### Bitwise Shift Expression Constraints

**Requirement: IDE-112: Bitwise Shift Expression Constraints (ver. 1)**

For a given bitwise shift expression  $e$  the following constraints must hold in N4JS mode: \* The types of the operands can be any.

$$\Gamma \vdash \text{BitwiseShiftExpression} \triangleleft \text{Expression:}\text{any}$$

## Type Inference

The type returned by a bitwise shift expression is always `number`:

$$\Gamma \vdash (\text{Expression} \ ( '<< '-' > '-' > > ' )\text{Expression}):\text{number}$$

A non-numeric operand is interpreted as 0, except for `true` which is interpreted as `1`; or objects implementing the symbol `toPrimitive`.

## 8.1.16. Relational Expression

### Syntax

Cf. [ECMA11a(p.p.77ff)]

```
RelationalExpression returns Expression: ShiftExpression
  (=> ({RelationalExpression.lhs=current} op=RelationalOperator) rhs=ShiftExpression)*;

RelationalExpressionNoIn returns Expression: ShiftExpression
  (=> ({RelationalExpression.lhs=current} op=RelationalOperatorNoIn) rhs=ShiftExpression)*;

enum RelationalOperator:
  lt='<' | gt='>' | lte='<=' | gte='>=' | instanceof | in;
RelationalOperatorNoIn returns RelationalOperator:
  '<' | '>' | '<=' | '>=' | 'instanceof';
```

### Semantics

#### Greater/Less (Equals) Operator Constraints

**Requirement: IDE-113:** Greater/Less (Equals) Operator Constraints (ver. 1)

For a given relational expression  $e$  with  $e.op \in \{<, >, <=, >=\}$  in N4JS mode, the following constraints must hold:

- The operands must have the same type and the type must be either a number, string, or boolean:

$$\begin{aligned} \Gamma \vdash rhs:T \quad T \in \{\text{number}, \text{string}, \text{boolean}\} \\ \Gamma \vdash lhs ('<' <= ' > =')rhs \triangleleft lhs:T \\ \Gamma \vdash rhs:O \quad O \notin \{\text{number}, \text{string}, \text{boolean}\} \quad T = \text{union}\{\text{number}, \text{string}, \text{boolean}\} \\ \Gamma \vdash lhs ('<' <= ' > =')rhs \triangleleft lhs:T \\ \Gamma \vdash lhs:T \quad T \in \{\text{number}, \text{string}, \text{boolean}\} \\ \Gamma \vdash lhs ('<' <= ' > =')rhs \triangleleft rhs:T \\ \Gamma \vdash lhs:O \quad O \notin \{\text{number}, \text{string}, \text{boolean}\} \quad T = \text{union}\{\text{number}, \text{string}, \text{boolean}\} \\ \Gamma \vdash lhs ('<' <= ' > =')rhs \triangleleft rhs:T \end{aligned}$$

#### Instanceof Operator Constraints

**Requirement: IDE-114:** Instanceof Operator Constraints (ver. 1)

For a given relational expression  $e$  with  $e.op = \text{instanceof}$ , the following constraints must hold:

- The right operand of the instanceof operator must be a `Function`<sup>1</sup>

In other words,

$$\Gamma \vdash lhs \ 'instanceof' \ rhs \triangleleft rhs:\text{type}\{\text{Class}\}$$

is contained in the the first type rule, an object type reference<sup>2</sup> or an enum type reference.

<sup>1</sup> Only `Function` implements the ECMAScript specification property `hasInstance`. Thus instanceof expressions are rewritten by the compiler for other types. Note that a reference to a class returns the constructor type, which actually is a function itself.

<sup>2</sup> Includes interfaces, since an interface type reference is a subtype of object type reference: `type{Interface} <: type{Object}`

$$\frac{\Gamma \vdash \text{lhs} \text{ 'instanceof' rhs} \triangleleft \text{rhs:Function}}{\Gamma \vdash \text{lhs} \text{ 'instanceof' rhs} \triangleleft \text{rhs:type[Object]}}$$

$$\frac{\Gamma \vdash \text{lhs} \text{ 'instanceof' rhs} \triangleleft \text{rhs:type[N4Enum]}}$$

The type of a definition site structural classifier  $C$  is not of type `C`. Thus, the `instanceof` operator cannot be used for structural types. Use-site structural typing is also not possible since `~` would be interpreted (by the parser) as a binary operator.

### Operator Constraints

#### Requirement: IDE-115: Operator Constraints (ver. 1)

For a given relational expression  $e$  with  $e.op = \text{in}$ , the following constraints must hold:

1. The right operand of the `in` operator must be an `Object`:

$$\Gamma \vdash \text{lhs} \text{ 'in' rhs} \triangleleft \text{rhs:Object}$$

2. In N4JS mode, the left operand is restricted to be of type `string` or `number`:

$$\Gamma \vdash \text{lhs} \text{ 'in' rhs} \triangleleft \text{lhs:union\{string, number\}}$$

A special feature of N4JS is support for interface type references in combination with the `instance of` operator. The compiler rewrites the code to make this work.

#### Example 8.14. `instanceof` with Interface

The following example demonstrates the use of the operator with an interface. This is, of course, not working in pure ECMAScript.

```
interface I {}

class A implements I {}
class B extends A {}
class C {}

function f(name: string, p: any) {
    if (p instanceof I) {
        console.log(name + " is instance of I");
    }
}

f("A", new A())
f("B", new B())
f("C", new C())
```

This will print out

```
A is instance of I
B is instance of I
```

### Type Inference

The type of a relational expression always is `boolean`:

$$\Gamma \vdash \text{lhs} \text{ ('i'|'i='|'i|'i='|'instanceof'|'in') rhs :boolean}$$

### 8.1.17. Equality Expression

#### Syntax

Cf. [ECMA11a(p.p.80ff)]

```
EqualityExpression returns Expression: RelationalExpression
(=> ({EqualityExpression.lhs=current} op=EqualityOperator) rhs=RelationalExpression)*;

EqualityExpressionNoIn returns Expression: RelationalExpressionNoIn
(=> ({EqualityExpression.lhs=current} op=EqualityOperator) rhs=RelationalExpressionNoIn)*;
```

```
enum EqualityOperator: same='===' | nsame='!==' | eq='==' | neq='!=';
```

## Semantics

There are no hard constraints defined for equality expressions.

In N4JS mode, a warning is created if for a given expression  $lhs('===' - '!==')rhs$ , neither  $\Gamma \vdash lhs.upper < :rhs.upper$  nor  $\Gamma \vdash rhs.upper < :lhs.upper$  and no interface or composed type is involved as the result is constant in these cases.

Note that a warning is only created if the upper bounds do not match the described constraints. This is necessary for wildcards. For example in

```
// with
class A{} class B extends A{}
function isFirst(ar: Array<? extends A>, b: B): boolean {
    return b === ar[0]
}
```

the type of array elements is `? extends A`.

Neither `? extends A < :B` nor `B < :? extends A` is true. This is why the upper bounds are to be used.

## Type Inference

The inferred type of an equality expression always is `boolean`.

$\overline{\Gamma \vdash lhs('==' | '!=')rhs :boolean}$

## 8.1.18. Binary Bitwise Expression

### Syntax

Cf. [ECMA11a(p.p.82ff)]

```
BitwiseANDExpression returns Expression: EqualityExpression
(=> ({BitwiseANDExpression.lhs=current} '&') rhs=EqualityExpression)*;

BitwiseANDExpressionNoIn returns Expression: EqualityExpressionNoIn
(=> ({BitwiseANDExpression.lhs=current} '&') rhs=EqualityExpressionNoIn)*;

BitwiseXORExpression returns Expression: BitwiseANDExpression
(=> ({BitwiseXORExpression.lhs=current} '^') rhs=BitwiseANDExpression)*;

BitwiseXORExpressionNoIn returns Expression: BitwiseANDExpressionNoIn
(=> ({BitwiseXORExpression.lhs=current} '^') rhs=BitwiseANDExpressionNoIn)*;

BitwiseORExpression returns Expression: BitwiseXORExpression
(=> ({BitwiseORExpression.lhs=current} '|') rhs=BitwiseXORExpression)*;

BitwiseORExpressionNoIn returns Expression: BitwiseXORExpressionNoIn
(=> ({BitwiseORExpression.lhs=current} '|') rhs=BitwiseXORExpressionNoIn)*;
```

## Semantics

### Bitwise Bitwise Expression Constraints

#### Requirement: IDE-116: Bitwise Bitwise Expression Constraints (ver. 1)

For a given bitwise bitwise expression  $e$  the following constraints must hold in N4JS mode:

- The types of the operands must be both number.

$\overline{\Gamma \vdash \text{BitwiseBitwiseExpression} \triangleleft \text{Expression: number}}$

## Type Inference

The type returned by a binary bitwise expression is always `number`.

## 8.1.19. Binary Logical Expression

### Syntax

```

LogicalANDExpression returns Expression: BitwiseORExpression
  (=> ({LogicalANDExpression.lhs=current} '&&') rhs=BitwiseORExpression)*;
LogicalANDExpressionNoIn returns Expression: BitwiseORExpressionNoIn
  (=> ({LogicalANDExpression.lhs=current} '&&') rhs=BitwiseORExpressionNoIn)*;

LogicalORExpression returns Expression: LogicalANDExpression
  (=> ({LogicalORExpression.lhs=current} '||') rhs=LogicalANDExpression)*;
LogicalORExpressionNoIn returns Expression: LogicalANDExpressionNoIn
  (=> ({LogicalORExpression.lhs=current} '||') rhs=LogicalANDExpressionNoIn)*;

```

## Semantics

### Binary Logical Expression Constraints

#### Requirement: IDE-117: Binary Logical Expression Constraints (ver. 1)

For a given binary logical expression  $e$  with  $e.lhs.type:L$  and  $e.rhs.type:R$  the following constraints must hold:

- In N4JS mode  $L$  must not be `undefined` or `null`.

## Type Inference

The evaluation relies on ECMAScript's abstract operation `ToBoolean` [ECMA11a(p.p.43)]. A short-circuit evaluation strategy is used so that depending on the types of the operands, different result types may be inferred. In particular, the inferred type usually is not `boolean` (cf. [ECMA11a(p.S11.11., p.p.83ff)]). The type inference does not take this short-circuit evaluation strategy into account, as it will affect the result in case one of the types is `null` either or `undefined`, which is not allowed in N4JS mode.

$$\Gamma \vdash lhs' \& \& ' - ' -- 'rhs : union\{\Gamma \vdash lhs, \Gamma \vdash rhs\}$$

## 8.1.20. Conditional Expression

### Syntax

Cf. [ECMA11a(p.S11.12, p.p.84)]

```

ConditionalExpression returns Expression: LogicalORExpression
  (=> ({ConditionalExpression.expression=current} '?') trueExpression=AssignmentExpression `;` 
       falseExpression=AssignmentExpression)?;

ConditionalExpressionNoIn returns Expression: LogicalORExpressionNoIn
  (=> ({ConditionalExpression.expression=current} '?') trueExpression=AssignmentExpression `;` 
       falseExpression=AssignmentExpressionNoIn)?;

```

## Semantics

### Conditional Expression Constraints

#### Requirement: IDE-118: Conditional Expression Constraints (ver. 1)

For a given conditional expression  $e$  with

$e.expression.type:C$ ,  
 $e.trueExpression.type:T$ ,  
 $e.falseExpression.type:F$

the following constraints must hold:

- A warning will be issued in N4JSmode if  $e.expression$  evaluates to a constant value. That is to say  $e.expression \in \{false, true, null, undefined\}$  or  $C \in \{void, undefined\}$ .

There are no specific constraints defined for the condition. The ECMAScript operation `ToBoolean` [ECMA11a(p.S9.2, p.p.43)] is used to convert any type to boolean.

## Type Inference

The inferred type of a conditional expression is the union of the true and false expression (cf. [ECMA11a(p.S11.12, p.p.84)]):

$$\frac{T = union\{\Gamma \vdash et, \Gamma \vdash ef\}}{\Gamma \vdash cond '?'et ':ef:T}$$

**Example 8.15. Type of Conditional Expressions**

Given the following declarations:

```
class A{}      class B extends A{}
class C{}      class D extends A{}
class G<T> { field: T; }

var ga: G<A>, gb: G<B>;
  a: A, b: B, c: C, d: D;
var boolean cond;
```

Then the type of the following conditional expression is inferred as noted in the comments:

```
cond ? a : a;          // A
cond ? a : b;          // union{A,B}
cond ? a : c;          // union{A,C}
cond ? b : d;          // union{B,D}
cond ? (cond ? a : b) : (cond ? c : d); // union{A,B,C,D}
cond ? (cond ? a : b) : (cond ? b : d); // union{A,B,D}
cond ? ga : gb;        // union{G<A>,G<B>}
```

## 8.1.21. Assignment Expression

### Syntax

```
AssignmentExpression <In, Yield>:
  lhs=Expression op=AssignmentOperator rhs=AssignmentExpression<In,Yield>
;
AssignmentOperator:
  '='
  '|=' | '/=' | '%=' | '+=' | '-='
  '|<<=' | '|>=' | '|>>='
  '|&=' | '|^=' | '|='
;
```

### Semantics

#### Simple Assignment

**Requirement: IDE-119: Simple Assignment (ver. 1)**

For a given assignment *assignment* with

*assignment.op* = '='

the following constraints must hold:

1. **[assignment.lhs] < : [assignment.rhs]**

In the following inference rule and the constraint, '@' is to be replaced with the right part of one of the assignment operators listed above, that is,

#### Compound Assignment

**Requirement: IDE-120: Compound Assignment (ver. 1)**

For a given assignment *left op right*, with *op* = '@=' but not '+=' or '-=', both, left and right must be subtypes of `number`.

For operator '+=' ,

- if the left-hand side is a `number`, then *left* '+' *right* must return a number as well. The right-hand side must, in fact, be a `number` (and not a `boolean`) here in order to avoid unexpected results.
- if the left-hand side is a `string`, then *left* '+' *right* must return a string as well. That means that the right-hand side can be of `any` type.

The expected type for the left-hand side is `union{number,string}`.

The basic idea behind these constraints is that the type of the left-hand side is not to be changed by the compound assignment.

## Write Access

### Requirement: IDE-121: Write Access (ver. 1)

For a given assignment expression *assignExpr*, the left-hand side must be writeable or a final data field and the assignment must be in the constructor. Let *v* be the bound variable (or field) with *bind(assignExpr.left, v)*

$$\begin{aligned} & v.\text{writeable} \vee v.\text{final} \wedge \quad v.\text{expr} = \text{null} \\ & \wedge \text{assignExpr.containingFunction} = v.\text{owner.constructor} \\ & \wedge \mu(\text{assignExpr.left}) = \text{PropertyAccess} \\ & \wedge \text{assignExpr.left.target} = "this" \end{aligned}$$

The value of *writeable* is true for setters and usually for variables and data fields. Assignability of variables and data fields can be restricted via `const` or the `@Final` annotation. See [Section 5.2.6.3, “Assignment Modifiers”](#)(data fields) and [Section 9.2.2, “Const”](#) (const variables) for details.

Also see [\[Req-IDE-93\]](#) for read access constraint.

The left-hand side of an assignment expression may be an array or object literal and the assignment expression is then treated as a destructuring assignment. See [Section 11.1, “Array and Object Destructuring”](#) for details.

## Type Inference

Similarly to [\[ECMA11a\(p.S11.1, p.p.84ff\)\]](#), we define type inference for simple assignment (`=`) and compound assignment (`op=`) individually.

The type of the assignment is simply the type of the right-hand side:

$$\frac{\Gamma \vdash \text{right}:T}{\Gamma \vdash \text{left} \ '=\text{right}:T}$$

Compound assignments are reduced to the former by splitting an operator `@=`, in which `@` is a simple operator, into a simple operator expression with operator `@` and a simple assignment `=`. Since the type of the latter is the right-hand side, we can define:

$$\frac{\Gamma \vdash \text{left} \ '@\text{right}:T}{\Gamma \vdash \text{left} \ '@=\text{right}:T}$$

### 8.1.22. Comma Expression

#### Syntax

Cf. [\[ECMA11a\(p.S11.14, p.p.85\)\]](#)

```
CommaExpression <In, Yield>:
    exprs+=AssignmentExpression<In,Yield> ',' exprs+=AssignmentExpression<In,Yield>
    (' ')   exprs+=AssignmentExpression<In,Yield>)*
;
```

#### Semantics

All expressions will be evaluated even though only the value of the last expression will be the result.

#### Example 8.16. Comma Expression

Assignment expressions precede comma expressions:

```
var b: boolean;
b = (12, 34, true); // ok, b=true
b = 12, 34, true ; // error, b=12 is invalid
```

## Type Inference

Cf. [\[ECMA11a\(p.S11.14, p.p.85\)\]](#)

The type of a comma expression *cexpr* is inferred to the last expression:

$$\frac{n=|\text{cexpr.exprs}|, \Gamma \vdash \text{cexpr.exprs}_n:T}{\Gamma \vdash \text{cexpr}:T}$$

## 8.2. ECMAScript 6 Expressions

### 8.2.1. The super Keyword

```
SuperLiteral: {SuperLiteral} 'super';
```

Apart from the use of keyword `super` in wildcards of type expressions (cf. [Section 4.2, “Type Expressions”](#)), there are two use cases for keyword `super`: super member access and super constructor calls.

#### Example 8.17. Super Keyword

Two use cases for keyword `super`:

```
class B extends A {
    constructor() {
        // super call
        super();
    }
    @Override
    m(): void {
        // super member access
        super.m();
    }
}
```

### Semantics

`super` can be used to access the supertype's constructor, methods, getters and setters. The supertype is defined lexically, which is different from how `this` works.<sup>3</sup>

Note that in [\[ECMA15a\] Chapter 12.3.5 The Super Keyword](#), `super` is defined as a keyword but the syntax and semantics are defined in conjunction of member access.

#### Type of Super is Always Nominal

##### Requirement: IDE-122: Type of Super is Always Nominal (ver. 1)

The type referenced with the super literal is always nominal. This is a consequence of references to types in extend clauses to be nominal.

$$\Gamma \vdash \text{super}: T \wedge T.\text{typingStrategy} = \text{nominal}$$

#### Access Super Constructor with Super Literal

##### Requirement: IDE-123: Access Super Constructor with Super Literal (ver. 1)

If the super literal  $s$  is used to access the super constructor of a class, all of the following constraints must hold:

1. The super constructor access must be a call expression:

$$\mu(cexpr) = \text{CallExpression} \wedge c.expr = cexpr$$

2. The super constructor call must be the expression of an expression statement  $exprStmt$ :

$$exprStmt = cexpr.container \wedge \mu(cexpr.container) = \text{ExpressionStatement}$$

3. The containing statement  $stmtExpr$  must be directly contained in a constructor body:

$$\mu(exprStmt.containingFunction) = \text{Constructor} \wedge exprStmt.container = exprStmt.containingFunction.body$$

4. There must be no access to and not return statement before the containing statement  $exprStmt$ .

Let  $si$  be the index of  $exprStmt$  in the constructor body:

$$exprStmt.container.stmts_{si} = exprStmt$$

<sup>3</sup> See [\[ECMA15a\]](#), Chapter 12.3.5 “The Super Keyword”; note the use of `HomeObject` instead of `thisValue`; also see this blog - <http://www.2ality.com/2011/11/super-references.html>.

Then, the following constraint must hold:<sup>4</sup>

$$\forall i < si \# \text{element} \in * \text{exprStmt.containerstmts}_i \quad \mu(i) \in \text{ThisLiteral, ReturnStatement}$$

Further constraints with regard to super constructor calls are described in [Section 4.11, “Constructor and Classifier Type”](#).

### Access Super Member with Super Literal

#### **Requirement: IDE-124: Access Super Member with Super Literal (ver. 1)**

If the super literal  $s$  is used to access a member of the super class, all of the following constraints must hold, with  $c = s.container.container$

1. The super literal must be the receiver of a method call (cf. remarks below):

$$\begin{aligned} \mu(c) &= \text{CallExpression} \\ \wedge \quad c.target &= \text{PropertyAccessExpression} \\ \wedge \quad c.target.target &= s \end{aligned}$$

2. The super literal is used in a method or field accessor of a class:

$\mu(s.containingClass) = \text{Class}$  3. The super literal must not be used in a nested function expression:

+  $\mu(s.containingFunction) = s.containingMethodOrFieldAccessor$  4. If the return type of the method access via super is this, the actually bound this type will be the type of the calling class (and not of the class defining the method).

+

$$\begin{aligned} s.containingClass &= T \quad \mu(m) = \text{Method} \quad m.returnType = \text{this} \\ &\quad \text{function}(): T < : s.m \end{aligned}$$

### Super Literal Usage

#### **Requirement: IDE-125: Super Literal Usage (ver. 1)**

For super literals, either [\[Req-IDE-123\]](#) or [\[Req-IDE-124\]](#) must hold, no other usage is allowed.

Consequences:

- Since fields cannot be overridden (except for changing the access modifier), it is not possible nor allowed to access a field via `super`.
- Super literals must not be used with index access (e.g., `super["foo"]`)
- It is not possible to chain super keywords. That is, it is not possible to call `super.super.m()`.
- It is not allowed to use the super literal in interfaces or non-methods/accessors.
- Super cannot be used to call an overridden method of an implemented method from the overriding method in the implementing class.
- In order to be able to access a super method of a method  $M$  of a class  $C$ , exactly one non-abstract super method  $M'$  in a super class  $S$  of  $C$  must exist. This is assured by the standard rules for binding identifiers.

If super is used to access a super member, the receiver type is not changed. This is important in particular for static methods as demonstrated in the following example:

#### **Example 8.18. Super Call in Static Methods**

```
class A {
    static foo(): void { console.log("A") }
    static bar(): void {
        this.foo();
    }
}

class B extends A {

    @Override
    static foo(): void { console.log("B") }
    @Override
```

<sup>4</sup>  $e \in^* c$  is the transitive version of  $e \in c$ , that is, it  $e$  directly or indirectly contained in  $c$ .

```

static bar(): void {
    A.bar();          ①
    super.bar();     ②
}
}

B.bar();

```

- ① The receiver (which is similar to the this-binding in ECMAScript) is changed to **A**.
- ② Using super, the receiver is preserved, i.e. **B**.

## 8.3. ECMAScript 7 Expressions

### 8.3.1. Await Expression

In N4JS, `await` is implemented as a unary operator with the same precedence as `yield` in ECMAScript 6.

Constraints governing the use of `await` are given together with those for `async` in [Section 6.4.1, “Asynchronous Functions”](#).

## 8.4. N4JS Specific Expressions

### 8.4.1. Class Expression

A class expression in N4JS is similar to a class expression in ECMAScript 6 [[ECMA15a\(p.14.5\)](#)].



Class expressions are not part of version 0.3

#### Syntax

See [Section 5.1.3, “Classes”](#).

#### Semantics and Type Inference

The inferred type of a class expression simply is the class type as described in [Section 4.11, “Constructor and Classifier Type”](#).

### 8.4.2. Cast (As) Expression

#### Syntax

```

CastExpression <Yield> returns Expression: expression=Expression 'as' targetTypeRef=TypeRefForCast;

TypeRefForCast returns StaticBaseTypeRef:
    ParameterizedTypeRef
    | ThisTypeRef
    | ConstructorTypeRef
    | ClassifierTypeRef
    | FunctionTypeExpression
    | UnionTypeExpression
    | IntersectionTypeExpression

```

### 8.4.3. Semantics and Type Inference

The inferred type of the type cast expression is the target type:

$\Gamma \vdash \text{expr} \text{ "as" } T : T$

The type cast returns the expression without further modifications. Type casts are simply removed during compilation so there will be no exceptions thrown at the cast until later when accessing properties which may not be present in case of a failed cast.

An error is issued if the cast is either unnecessary or cannot succeed. See further details in [Section 7.3.1, “Type Cast”](#).

## 8.5. Compile-Time Expressions

A compile-time expression is an expression that can be fully evaluated at compile time. Not all expressions introduced in the previous sections qualify as compile-time expressions. Some forms of expressions always qualify (e.g. a string literal is always a compile-

time expression), some never (e.g. call expressions), and for some expressions the operands must be of a certain value. The latter applies, for example, to division: `5 / 0` is a valid ECMAScript expression (evaluating to `NaN`) but is not a compile-time expression. So it's the actual compile-time value of the divisor that makes the difference, here. In any case, if an expression has operands, it is a compile-time expression only if all operands are compile-time expressions.

The value a compile-time expression evaluates to at compile-time is called *compile-time value*. So, an expression has a compile-time value if and only if it is a compile-time expression.

## Compile-Time Expression

### Definition: Compile-Time Expression

The following expressions are called compile-time expressions:

- `undefined` (but not `NaN` or `Infinity`).
- the `null` literal.
- all boolean, numeric, and string literals.
- template string literals, iff all embedded expressions are compile-time expressions.
- a parenthesis expression, iff its nested expression is a compile-time expression.
- unary expressions in case of the following operators:
  - `!` iff the operand is a compile-time expression and evaluates to a boolean value.
  - `+` iff the operand is a compile-time expression and evaluates to a numeric value.
  - `-` iff the operand is a compile-time expression and evaluates to a numeric value.
  - `void`.
- binary expressions in case of the following operators:
  - `+` iff both operands are compile-time expressions and
    - both evaluate to numeric values, or
    - at least one evaluates to a string value.
  - `-`, `*` iff both operands are compile-time expressions and evaluate to numeric values.
  - `/`, `%` iff both operands are compile-time expressions and evaluate to numeric values and the right-hand operand is non-zero (i.e. division by zero is disallowed in compile-time expression, because `NaN` is not a supported compile-time value).
  - `&&`, `||` iff both operands are compile-time expressions and evaluate to boolean values.
- a tertiary conditional expression, iff the first operand is a compile-time expression evaluating to a boolean value `B` and
  - in case `B` is true, the second operand is a compile-time expression.
  - in case `B` is false, the third operand is a compile-time expression.
- an identifier reference to a `const` variable, iff its initializer expression is a compile-time expression.
- a property access expression, iff it is direct (see [the section called "Direct Property Access"](#)) and refers to
  - a built-in symbol, e.g. `Symbol.iterator`,
  - a literal of a `@stringBased` enum, or
  - a `const` field with a compile-time initializer expression.

In all other cases, the expression is not a compile-time expression.

Every expression in the code may be a compile-time expression, but in most places this has no particular effect and is simply ignored. They are of significance only in computed property names, in index access expressions, as initializers of `const` variables and fields (as stated above) and when nested as an operand inside an expression at these locations.



# Chapter 9. Statements

For all statements, we define the following pseudo properties:

`containingFunction` The function or method in which the statement is (indirectly) contained, this may be null.

`containingClass` The class in which the statement is (indirectly) contained, this may be null.

The expressions and statements are ordered, at first describing the constructs available in the 5th edition of ECMA-262 referred to as [ECMA11a] in the following. The grammar snippets already use newer constructs in some cases.

## 9.1. ECMAScript 5 Statements

N4JS supports the same statements as ECMAScript. Some of these statements are enhanced with annotations [Chapter 10, Annotations](#) and type information.

Although some statements may return a value which can be used via certain constructs such as `eval` ), no type is inferred for any statement. The compiler will always create a warning if a statement is used instead of an expression.

The following sections, therefore, do not define how to infer types for statement but how types and type annotations are used in these statements and the specific type constraints for a given statement.

All syntax definitions taken from [ECMA11a] are repeated here for convenience reasons and in order to define temporary variables for simplifying constraint definitions. If non-terminals are not defined here, the definition specified in [ECMA11a] is to be used.

### 9.1.1. Function or Field Accessor Bodies

Dead Code

**Requirement: IDE-126: Dead Code (ver. 1)**

For all statements in a function or field accessor (getter/setter) body, the following constraints must hold:

1. Statements appearing directly after return, throw, break, or continue statements (in the same block) are considered to be dead code and a warning is issued in these cases.

### 9.1.2. Variable Statement

#### Syntax

A var statement can declare the type of the variable with a type reference. This is described with the following grammar similar to [ECMA11a(p.S12.2, p.p.87)]:

```
VariableStatement <In, Yield>:
    => ({VariableStatement}
        'var'
    )
    varDeclsOrBindings+=VariableDeclarationOrBinding<In,Yield,false> (',')
    varDeclsOrBindings+=VariableDeclarationOrBinding<In,Yield,false>)* Semi
;

VariableDeclarationOrBinding <In, Yield, OptionalInit>:
    VariableBinding<In,Yield,OptionalInit>
    | VariableDeclaration<In,Yield,true>
;

VariableBinding <In, Yield, OptionalInit>:
    => pattern=BindingPattern<Yield> (
        <OptionalInit> ('=' expression=AssignmentExpression<In,Yield>)?
        | <OptionalInit> '=' expression=AssignmentExpression<In,Yield>
    )
;

VariableDeclaration <In, Yield, AllowType>:
    {VariableDeclaration} VariableDeclarationImpl<In,Yield,AllowType>;
;

fragment VariableDeclarationImpl <In, Yield, AllowType>*:
    annotations+=Annotation*
```

```

(
  <AllowType> =>
    name=BindingIdentifier<Yield> ColonSepTypeRef?
  ) ('=' expression=AssignmentExpression<In,Yield>)?
| <!AllowType> =>
  name=BindingIdentifier<Yield>
  ) ('=' expression=AssignmentExpression<In,Yield>)?
)
;

```

**Example 9.1. Variable Statement**

```

var any: any;
// any.type := any

var anyNull = null;
// anyNull.type := any

var s: string;
// s.type := string

var init = "Hi";
// init.type := string

const MESSAGE = "Hello World";
// MESSAGE.type := string

```

**Semantics**

From a model and type inference point of view, variable and constant statements and declarations are similar except that the pseudo property `const` is set to false for variables and true for constants. Also see exported variable statement ([Section 9.2.5, “Export Statement”](#)) and constant statement and declaration ([Section 9.2.2, “Const”](#)).

**Variable declaration****Requirement: IDE-127: Variable declaration (ver. 1)**

For a given variable declaration  $d$ , the following constraints must hold:

- The type of the initializer expression must conform to the declared type:

$$d.expression \neq \text{null} \wedge d.declaredTypeRef \neq \text{null} \\ \rightarrow \Gamma \vdash d.expression:type < : \Gamma \vdash d.declaredTypeRef$$

- The initializer expression should not contain a reference to  $d$  except where the reference is contained in a class expression or function expression and the class is not immediately initialized or the function is not immediately invoked. In these cases, the code is executed later and the self-reference is not a problem.

To clarify: *should not* means that only a warning will be produced.

```

// not ok (simple case)
var n = n + 1;

// ok (class expression not fully supported)
// var cls1 = class { static sfield1 = "hello"; field2 = cls1.sfield1; };

// not ok, immediately instantiated (class expression not fully supported)
// var cls2 = new class { field1 = "hello"; field2 = cls2.field1; };

// ok
var fun1 = function() : number { var x = fun1; return -42; };

// not ok, immediately invoked
var fun2 = function() : number { var x = fun2; return -42; }();

```

The variable statement may contain array or object destructuring patterns, see [Section 11.1, “Array and Object Destructuring”](#) for details.

**Type Inference**

The type of a variable is the type of its declaration:

The type of a variable declaration is either the declared type or the inferred type of the initializer expression:

### 9.1.3. If Statement

#### Syntax

Cf. [ECMA11a(p.S12.5, p.p.89)]

```
IfStatement <Yield>:
  'if' '(' expression=Expression<In=true,Yield> ')'
  ifStmt=Statement<Yield>
  (=> 'else' elseStmt=Statement<Yield>)?;
```

#### Semantics

There are no specific constraints defined for the condition, the ECMAScript operation `ToBoolean` [ECMA11a(p.S9.2, p.p.43)] is used to convert any type to boolean.

#### If Statement

**Requirement: IDE-128: If Statement (ver. 1)**

In N4JS, the expression of an if statement must not evaluate to `void`. If the expression is a function call in particular, the called function must not be declared to return `void`.

### 9.1.4. Iteration Statements

#### Syntax

Cf. [ECMA11a(p.S12.6, p.p.90ff)]

The syntax already considers the for-of style described in Section 9.2.3, “`for ... of` statement”.

```
IterationStatement <Yield>:
  DoStatement<Yield>
  | WhileStatement<Yield>
  | ForStatement<Yield>
;

DoStatement <Yield>: 'do' statement=Statement<Yield> 'while' '(' expression=Expression<In=true,Yield> ')' => Semi?;
WhileStatement <Yield>: 'while' '(' expression=Expression<In=true,Yield> ')' statement=Statement<Yield>;

ForStatement <Yield>:
  {ForStatement} 'for' '('
  (
    // this is not in the spec as far as I can tell, but there are tests that rely on this to be valid JS
    =>(initExpr=LetIdentifierRef forIn?='in' expression=Expression<In=true,Yield> ')')
  | ( ->varStmtKeyword=VariableStatementKeyword
    (
      =>(varDeclsOrBindings+=BindingIdentifierAsVariableDeclaration<In=false,Yield> (forIn?='in' | forOf?='of'))
      ->expression=AssignmentExpression<In=true,Yield>?
      | varDeclsOrBindings+=VariableDeclarationOrBinding<In=false,Yield,OptionalInit=true>
        (
          (, varDeclsOrBindings+=VariableDeclarationOrBinding<In=false,Yield,false>)* ')'
        expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
        | forIn?='in' expression=Expression<In=true,Yield>?
        | forOf?='of' expression=AssignmentExpression<In=true,Yield>?
      )
    )
  | initExpr=Expression<In=false,Yield>
  (
    ';' expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
    | forIn?='in' expression=Expression<In=true,Yield>?
    | forOf?='of' expression=AssignmentExpression<In=true,Yield>?
  )
  | ';' expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
)
'
) statement=Statement<Yield>;
;

ContinueStatement <Yield>: {ContinueStatement} 'continue' (label=[LabelledStatement/BindingIdentifier<Yield>])? Semi;
BreakStatement <Yield>: {BreakStatement} 'break' (label=[LabelledStatement/BindingIdentifier<Yield>])? Semi;
```

Since `varDecl(s)` are `variableStatement`s as described in Section 9.1.2, “Variable Statement”, the declared variables can be type annotated.



Using for-in is not recommended, instead `_each` should be used.

## Semantics

There are no specific constraints defined for the condition, the ECMAScript operation `ToBoolean` [ECMA11a(p.S9.2, p.p.43)] is used to convert any type to boolean.

### For-In-Statement Constraints

#### Requirement: IDE-129: For-In-Statement Constraints (ver. 1)

For a given  $f$  the following conditions must hold:

- The type of the expression must be conform to object:  
 $\Gamma \vdash f.expression < :unionObject, string, ArgumentType$
- Either a new loop variable must be declared or an rvalue must be provided as init expression:  
 $f.varDecl \neq null \vee (f.initExpr \neq null \wedge isRValue(f.initExpr))$
- The type of the loop variable must be a string (or a super type of string, i.e. any):  
 $(f.varDecl \neq null \wedge \Gamma \vdash f.varDecl < :string)$   
 $\vee (f.initExpr \neq null \wedge \Gamma \vdash string < :f.initExpr)$

## 9.1.5. Return Statement

### Syntax

The returns statement is defined as in [ECMA11a(p.S12.9, p.p.93)] with

```
ReturnStatement <Yield>:
  'return' (expression=Expression<In=true,Yield>)? Semi;
```

## Semantics

### Return statement

#### Requirement: IDE-130: Return statement (ver. 1)

- Expected type of expression in a return statement must be a sub type of the return type of the enclosing function:

+

Note that the expression may be evaluated to `void`. 2. If enclosing function is declared to return `void`, then either \* no return statement must be defined \* return statement has no expression \* type of expression of return statement is `void`  
 3. If enclosing function is declared to to return a type different from `void`, then \* all return statements must have a return expression \* all control flows must either end with a return or throw statement 4. Returns statements must be enclosed in a function. A return statement, for example, must not be a top-level statement.

## 9.1.6. With Statement

### Syntax

The with statement is not allowed in N4JS, thus an error is issued.

```
WithStatement <Yield>:
  'with' '(' expression=Expression<In=true,Yield> ')'
  statement=Statement<Yield>;
```

## Semantics

N4JS is based on strict mode and the with statement is not allowed in strict mode, cf. [ECMA11a(p.S12.10.1, p.p.94)].

### With Statement

**Requirement: IDE-131: With Statement (ver. 1)**

With statements are not allowed in N4JS or strict mode.

### 9.1.7. Switch Statement

#### Syntax

Cf. [ECMA11a(p.S12.11, p.p.94ff)]

```
switchStatement <Yield>:
    'switch' '(' expression=Expression<In=true,Yield> ')'
    ('cases'+=CaseClause<Yield>)*
    ((cases+=DefaultClause<Yield>)
     (cases+=CaseClause<Yield>)? ')'
;

CaseClause <Yield>: 'case' expression=Expression<In=true,Yield> ':' (statements+=Statement<Yield>)*;
DefaultClause <Yield>: {DefaultClause} 'default' ':' (statements+=Statement<Yield>)*;
```

#### Semantics

##### Switch Constraints

**Requirement: IDE-132: Switch Constraints (ver. 1)**

For a given switch statement  $s$ , the following constraints must hold:

- For all cases  $c \in s.cases$ ,  $s.expr == c.expr$  must be valid according to the constraints defined in [Section 8.1.17, “Equality Expression”](#).

### 9.1.8. Throw, Try, and Catch Statements

#### Syntax

Cf. [ECMA11a(p.S12.13/14, p.p.96ff)]

```
ThrowStatement <Yield>:
    'throw' expression=Expression<In=true,Yield> Semi;

TryStatement <Yield>:
    'try' block=Block<Yield>
    ((catch=CatchBlock<Yield> finally=FinallyBlock<Yield>?) | finally=FinallyBlock<Yield>)
;

CatchBlock <Yield>: {CatchBlock} 'catch' '(' catchVariable=CatchVariable<Yield> ')' block=Block<Yield>;

CatchVariable <Yield>:
    =>bindingPattern=BindingPattern<Yield>
    | name=BindingIdentifier<Yield>
;

FinallyBlock <Yield>: {FinallyBlock} 'finally' block=Block<Yield>;
```

There must be no type annotation for the catch variable, as this would lead to the wrong assumption that a type can be specified.

#### Type Inference

The type of the catch variable is always assumed to be `any`.

$\Gamma \vdash \text{catchBlock}.\text{catchVariable} : \text{any}$

### 9.1.9. Debugger Statement

#### Syntax

Cf. [ECMA11a(p.S12.15, p.p.97ff)]

```
DebuggerStatement: {DebuggerStatement} 'debugger' Semi;
```

## Semantics

na

## 9.2. ECMAScript 6 Statements

N4JS export and import statements are similar to ES6 with some minor differences which are elaborated on below.

### 9.2.1. Let

Cf. [ECMA11a(p.13.2.1)], also Rauschmayer, 2ality: [Variables and scoping in ECMAScript 6](#)

### 9.2.2. Const

Cf. [ECMA15a(p.13.2.1)], also Rauschmayer, 2ality: [Variables and scoping in ECMAScript 6](#)

Additionally to the `var` statement, the `const` statement is supported. It allows for declaring variables which must be assigned to a value in the declaration and their value must not change. That is to say that constants are not allowed to be on the left-hand side of other assignments.

```
ConstStatement returns VariableStatement: 'const' varDecl+=ConstDeclaration ( ',' varDecl+=ConstDeclaration )* Semi;
ConstDeclaration returns VariableDeclaration: typeRef=TypeRef? name=IDENTIFIER const?= '=' expression=AssignmentExpression;
```

## Semantics

A const variable statement is more or less a normal variable statement (see [Section 9.1.2, “Variable Statement”](#)), except that all variables declared by that statement are not writable (cf. [Req-IDC-121]). This is similar to constant data fields (cf. [Section 5.2.6.3, “Assignment Modifiers”](#)).

### Writability of const variables

**Requirement: IDE-133:** [Writability of const variables](#) (ver. 1)

All variable declarations of a const variable statement `constStmt` are not writeable:

$\forall vdecl \in constStmt.varDecl: \neg vdecl.writable$

### 9.2.3. `for ... of` statement

ES6 introduced a new form of `for` statement: `for ... of` to iterate over the elements of an `Iterable`, cf. [Section 4.8.3, “IterableN”](#).

## Syntax

See [Section 9.1.4, “Iteration Statements”](#)

## Semantics

### for ... of statement

**Requirement: IDE-134:** [for ... of statement](#) (ver. 1)

For a given  $f$  the following conditions must hold:

1. The value provided after `of` in a `for ... of` statement must be a subtype of `Iterable<?>`.
2. Either a new loop variable must be declared or an rvalue must be provided as init expression:  
 $f.varDecl \neq null \vee (f.initExpr \neq null \wedge isRValue(f.initExpr))$
3. If a new variable  $v$  is declared before `of` and it has a declared type  $T$ , the value provided after must be a subtype of `Iterable<?extends T>`. If  $v$  does not have a declared type, the type of  $v$  is inferred to the type of the first type argument of the actual type of the value provided after `of`.
4. If a previously-declared variable is referenced before with a declared or inferred type of  $T$ , the value provided after `of` must be a subtype of `Iterable<?extends T>`.



**Iterable** is structurally typed on definition-site so non-N4JS types can meet the above requirements by simply implementing the only method in interface **Iterable** (with a correct return type).



The first of the above constraints (the type required by the 'of' part in a **for ... of** loop is **Iterable**) was changed during the definition of ECMAScript 6. This is implemented differently in separate implementations and even in different versions of the same implementation (for instance in different versions of V8). Older implementations require an **Iterator** or accept both **Iterator** and **Iterable**.

Requiring an **Iterable** and not accepting a plain **Iterator** seems to be the final decision (as of Dec. 2014). For reference, see abstract operations **GetIterator** in [ECMA15a(p.S7.4.2)] and "CheckIterable" [ECMA15a(p.S7.4.1)] and their application in "ForIn/OfExpressionEvaluation" [ECMA15a(p.S13.6.4.8)] and **CheckIterable** and their application in **ForIn/OfExpressionEvaluation**. See also a related blog post <sup>1</sup> that is kept up to date with changes to ECMAScript 6:

*ECMAScript 6 has a new loop, for-of. That loop works with iterables. Before we can use it with createArrayIterator(), we need to turn the result into an iterable.*

An array or object destructuring pattern may be used left of the **of**. This is used to destructure the elements of the **Iterable** on the right-hand side (not the **Iterable** itself). For details, see Section 11.1, "Array and Object Destructuring".

## 9.2.4. Import Statement

Cf. ES6 import [ECMA15a(p.15.2.2)], see also <https://babeljs.io/docs/usage/modules/>

### Syntax

The grammar of import declarations is defined as follows:

```

ImportDeclaration:
  { ImportDeclaration}
  ImportDeclarationImpl
;

fragment ImportDeclarationImpl*:
  'import' (
    ImportClause importFrom?='from'
  )? module=[types::TModule/ModuleSpecifier] Semi
;

fragment ImportClause*:
  importSpecifiers+=DefaultImportSpecifier ('.' ImportSpecifiersExceptDefault)?
  | ImportSpecifiersExceptDefault
;

fragment ImportSpecifiersExceptDefault*:
  importSpecifiers+=NamespaceImportSpecifier
  | '{' (importSpecifiers+=NamedImportSpecifier ('.' importSpecifiers+=NamedImportSpecifier)* ','?)? '}'
;

NamedImportSpecifier:
  importedElement=[types::TExportableElement/BindingIdentifier<Yield=false>]
  | importedElement=[types::TExportableElement/IdentifierName] 'as' alias=BindingIdentifier<Yield=false>
;

DefaultImportSpecifier:
  importedElement=[types::TExportableElement/BindingIdentifier<Yield=false>]
;

NamespaceImportSpecifier: {NamespaceImportSpecifier} '*' 'as' alias=BindingIdentifier<false> (declaredDynamic?='+')?
;

ModuleSpecifier: STRING;

```

These are the properties of import declaration which can be specified by the user:

**annotations**

Arbitrary annotations, see Chapter 10, *Annotations* and below for details.

<sup>1</sup> available at: <http://www.2ality.com/2013/06/iterators-generators.html>

`importSpecifiers` The elements to be imported with their names.

Also see compilation as described in [Section 12.8, “Modules”](#), for semantics see [Import Statement Semantics](#).

#### Example 9.2. Import

```
import A from "p/A"
import {C,D,E} from "p/E"
import * as F from "p/F"
import {A as G} from "p/G"
import {A as H, B as I} from "p/H"
```

## Semantics

Import statements are used to import identifiable elements from another module. Identifiable elements are

- types (via their type declaration), in particular
  - classifiers (classes, interfaces)
  - functions
- variables and constants.

The module to import from is identified by the string literal following keyword `from`. This string must be a valid

- complete module specifier<sup>2</sup>:

```
import {A} from "ProjectA/a/b/c/M"
```

- plain module specifier:

```
import {A} from "a/b/c/M"
```

- or project name only, assuming the project defines a main module in its `package.json` file (using the `mainModule` package.json property, see [mainModule](#)):

```
import {A} from "ProjectA"
```

For choosing the element to import, there are the exact same options as in ECMAScript6:

- named imports select one or more elements by name, optionally introducing a local alias:

```
import {C} from "M"
import {D as MyD} from "M"
import {E, F as MyF, G, H} from "M"
```

- namespace imports select all elements of the remote module for import and define a namespace name; the imported elements are then accessed via the namespace name:

```
import * as N from "M"
var c: N.C = new N.C();
```

- default imports select whatever element was exported by the remote module as the default (there can be at most one default export per module):

```
import C from "M"
```

- namespace imports provide access to the default export:

```
import * as N from "M"
var c: N.default = new N.default();
```

The following constraints are defined on a (non-dynamic) import statement *i*:

- The imported module needs to be accessible from the current project.
- The imported declarations need to be accessible from the current module.

For named imports, the following constraints must hold:

<sup>2</sup> For more details on module specifiers, see [Section 3.2.3, “Qualified Names”](#).

- No declaration must be imported multiple times, even if aliases are used.
- The names must be unique in the module. They must not conflict with each other or locally declared variables, types, or functions.
- Declarations imported via named imports are accessible only via used name (or alias) and not via original name directly.

For wildcard imports, the following constraints must hold:

- Only one namespace import can be used per (target) module, even if different namespace name is used.
- The namespace name must be unique in the module. They must not conflict with each other or locally declared variables, types, or functions.
- Declarations imported via namespace import are accessible via namespace only and not with original name directly.

For namespace imports, the following constraints must hold:

- If the referenced module is a plain `js` file, a warning will be created to use the dynamic import instead.

For default imports, the following constraints must hold:

- The referenced module must have a default export.

Cross-cutting constraints:

- No declaration can be imported via named import and namespace import at the same time.

#### Example 9.3. Imports

Imports cannot be duplicated:

```
import * as A from 'A';
import * as A from 'A' //error, duplicated import statement

import B from 'B';
import B from 'B' //error, duplicated import statement
```

Given element cannot be imported multiple times:

```
import * as A1 from 'A';
import * as A2 from 'A' //error, elements from A already imported in A1

import B from 'B';
import B as B1 from 'B' //error, B/B is already imported as B

import C as C1 from 'C';
import C from 'C' //error, C/C is already imported as C1

import D as D1 from 'D';
import D as D2 from 'D' //error, D/D is already imported as D1

import * as NE from 'E';
import E from 'E' //error, E/E is already imported as NE.E

import F from 'F';
import * as NF from 'F' //error, F/F is already imported as F
```

Names used in imports must not conflict with each other or local declarations:

```
import * as A from 'A1';
import * as A from 'A2';//A is already used as namespace for A1

import B from 'B1';
import B1 as B from 'B2';//B us already used as import B/B1

import C1 as C from 'C1';
import * as C from 'C2'; //C is already used as import C1/C1

import * as D from 'D1';
import D2 as D from 'D2';//D is already used as namespace for D1

import E from 'E';
var E: any; // conflict with named import E/E

import * as F from 'F';
var F: any; // conflict with namespace F
```

Using named imports, aliases and namespaces allows to refer to multiple types of the same name such as `A/A`, `B/A` and `C/A`:

```
import A from 'A';// local name A references to A/A
import A as B from 'B';//local name B references to B/A
import * as C from 'C';//local name C.A references to C/A
```

If a declaration has been imported with an alias or namespace, it is not accessible via its original name:

```
import * as B from 'A1';
import A2 as C from 'A2';

var a1_bad: A1;//error, A1/A1 is not directly accessible with original name
var a1_correct: B.A1;// A1/A1 is accessible via namespace B
var a2_bad: A2;//error, A2/A2 is not directly accessible with original name
var a2_correct: C;// A2/A2 is accessible via alias C
```

### 9.2.4.1. Dynamic Imports

N4JS extends the ES6 module import in order that modules without a `n4jsd` or `n4js` file (plain `js` modules) can be imported. This is done by adding `+` to the name of the named import.

#### Dynamic Import

##### **Requirement: IDE-136: Dynamic Import (ver. 1)**

Let  $i$  be an import statement  $i$  with a dynamic namespace specifier. The following constraints must hold:

1.  $i.module$  must not reference an `n4js` file.
2. If  $i.module$  references an `n4jsd` file, a warning is to be created.
3. If the file referenced by  $i.module$  is not found, an error is created just as in the static case.

These constraints define the error level when using dynamic import: we receive no error for `js`, a warning for `n4jsd`, and an error for `n4js` files. The idea behind these distinct error levels is as follows:

If only a plain `js` file is available, using the dynamic import is the only way to access elements from the `js` module. This might be an unsafe way, but it allows the access and simplifies the first steps. An `n4jsd` file may then be made available either by the developer using the `js` module or by a third-party library. In this case, we do not want to break existing code.

There is only a warning created in the case of an available `n4jsd` file and a `js` file still must be provided by the user.

Having an `n4js` file is a completely different story; no `n4jsd` file is required, no `js` file is needed (since the transpiler creates one from the `n4js` file) and there is absolutely no reason to use the module dynamically.

### 9.2.4.2. Immutability of Imports

Imports create always immutable bindings, c.f. [ECMA15a(p.8.1.1.5)] <http://www.ecma-international.org/ecma-262/6.0/index.html#sec-createimportbinding>

#### Immutable Import

##### **Requirement: IDE-137: Immutable Import (ver. 1)**

Let  $i$  be a binding to an imported element. It is an error if

- $i$  occurs on the left-hand side as the assignment-target of an assignment expression (this also includes any level in a destructuring pattern on the left-hand side),
- $i$  is a direct argument of a postfix operator (`i++/i--`),
- $i$  is a direct argument of a `delete` operator,
- $i$  is a direct argument of the `increment` or `decrement` unary operator (`i++/i--`)

### 9.2.5. Export Statement

Cf. ES6 import [ECMA15a(p.15.2.3)]

#### Syntax

Grammar of export declarations is defined as follows:

```
ExportDeclaration:
  {ExportDeclaration}
  ExportDeclarationImpl
;
```

```

fragment ExportDeclarationImpl*:
  'export' (
    wildcardExport?='*' ExportFromClause Semi
    | ExportClause ->ExportFromClause? Semi
    | exportedElement=ExportableElement
    | defaultExport?='default' (->exportedElement=ExportableElement |
    defaultExportedExpression=AssignmentExpression<In=true,Yield=false> Semi)
  )
;

fragment ExportFromClause*:
  'from' reexportedFrom=[types: :TModule/ModuleSpecifier]
;

fragment ExportClause*:
  '{'
    (namedExports+=ExportSpecifier (',' namedExports+=ExportSpecifier)* ','?)?
  '}'
;

ExportSpecifier:
  element=IdentifierRef<Yield=false> ('as' alias=IdentifierName)?
;

ExportableElement:
  N4ClassDeclaration<Yield=false>
  | N4InterfaceDeclaration<Yield=false>
  | N4EnumDeclaration<Yield=false>
  | ExportedFunctionDeclaration<Yield=false>
  | ExportedVariableStatement
;

```

These are the properties of export declaration, which can be specified by the user:

**exportedElement** The element to be exported, can be a declaration or a variable/const statement.

#### Example 9.4. Export

```

export public class A{}
export interface B{}
export function foo() {}
export var a;
export const c="Hello";

```

## Semantics

With regard to type inference, export statements are not handled at all. Only the exported element is inferred and the `export` keyword is ignored.

In order to use types defined in other compilation units, these types have to be explicitly imported with an import statement.

Imported namespaces cannot be exported.

Declared elements (types, variables, functions) are usually only visible outside the declaring module if the elements are exported and imported (by the using module, cf. [Section 9.2.4, “Import Statement”](#)).

Some special components (runtime environment and libraries, cf. [Section 12.2.2, “Runtime Environment and Runtime Libraries”](#), may export elements globally. This is done by annotating the export (or the whole module) with `@Global`, see [Section 13.3, “Global Definitions”](#) for details.

By adding `default` after the keyword `export`, the identifiable element can be exported as ‘the default’. This can then be imported from other modules via default imports (see [Section 9.2.4, “Import Statement”](#)).

# Chapter 10. Annotations

## 10.1. Introduction

Annotations are used to further define meta properties of language elements such as types, variables and functions. These annotations are used by the compiler and validator to prohibit the developer from introducing constructs which are either not allowed or are unnecessary in certain contexts.

Since annotations are to be processed by the compiler and the compilation cannot be extended by third-party users for security reasons, annotations cannot be defined by developers. Instead, the compiler comes with a predefined set of annotations which are summarized here.

### 10.1.1. Syntax

Annotations are used similarly as in Java (although new annotations cannot be defined by the user). They are formally defined as follows:

```
Annotation: '@' AnnotationNoAtSign;
ScriptAnnotation returns Annotation: '@@' AnnotationNoAtSign;

AnnotationNoAtSign returns Annotation:
    name=AnnotationName (=> '(' (args+=AnnotationArgument (',' args+=AnnotationArgument)*)? ')')?;

AnnotationArgument:
    LiteralAnnotationArgument | TypeRefAnnotationArgument
;

LiteralAnnotationArgument:
    literal=Literal
;

TypeRefAnnotationArgument:
    typeRef=TypeRef
;
```

### 10.1.2. Properties

We use the map notation for retrieving annotation properties and values from a list of annotations, for example `x.annotations[Required]`, or shorter `x@Required`.

### 10.1.3. Element-Specific Annotations

The following annotations are element-specific and are explained in the corresponding sections:

**Table 10.1. Element-Specific Annotations**

Annotation	Element Types	Section
<code>@Internal</code>	<code>TypeDefiningElement, Member, Function, Export</code>	Section 3.1, “Access Control”
<code>@Undefined</code>	<code>Variable</code>	Section 4.5.1, “Undefined Type”
<code>@StringBased</code>	<code>Enum</code>	Section 4.13.2, “String-Based Enums”
<code>@Final</code>	<code>Class, Member</code>	Section 5.2.3.4, “Final Methods”
<code>@Spec</code>	<code>FPar</code>	[Req-IDE-59]
<code>@Override</code>	<code>Method</code>	Section 5.2.8, “Redefinition of Members”
<code>@Promisifiable</code>	<code>Function</code>	Section 6.5.2, “Promisifiable Functions”
<code>@Promisify</code>	<code>CallExpression</code>	Section 6.5.2, “Promisifiable Functions”
<code>@This</code>	<code>Function</code>	Section 2.1.2, “This Keyword”
<code>@N4JS</code>	<code>Class, Export Statement</code>	Section 13.2, “External Declarations”
<code>@IgnoreImplementation</code>	<code>Script, ExportDeclaration, ExportableElement</code>	Section 13.2, “External Declarations”
<code>@Global</code>	<code>External Declaration</code>	Section 13.3, “Global Definitions”
<code>@ProvidedByRuntime</code>	<code>External Declaration</code>	Section 13.4, “Runtime Definitions”
<code>@TestAPI</code>	<code>TypeDefiningElement, Member</code>	Section 11.3, “Test Support”
<code>@Polyfill</code>	<code>Class</code>	Section 11.4, “Polyfill Definitions”
<code>@StaticPolyfill</code>	<code>Class</code>	Section 11.4.2, “Static Polyfill Definitions”
<code>@StaticPolyfillAware</code>	<code>Script</code>	Section 11.4.2, “Static Polyfill Definitions”
<code>@StaticPolyfillModule</code>	<code>Script</code>	Section 11.4.2, “Static Polyfill Definitions”
<code>@Transient</code>	<code>Field</code>	a

<sup>a</sup>intended for internal use only; will be removed

### 10.1.4. General Annotations

#### 10.1.4.1. IDEBUG

`@IDEBUG` is an annotation similar to Java’s `@SuppressWarnings`. It changes the severity of an issue from an error to a warning so that code can be compiled regardless of validation errors. This is to be used for known IDE bugs only.

#### 10.1.5. Syntax

```
'@IDEBUG' '(' bugID = INT ',' errorMessage=STRING ')'
```

The annotation is defined transitively and repeatable on script, type declaration, function and method level.

#### 10.1.5.1. Semantics

This annotation will cause errors issued in the scope of the annotation (in the defined script, type, or method) to be transformed to warnings if their message text is similar to the `errorMessage` text. If `errorMessage` ends with ... (three dots as a single character, created by Eclipse to abbreviate messages), then the error’s message text must start with the specified text.

If no matching error is found, the annotation itself will issue an error.

##### Example 10.1. IDEBUG Example

In the following code snippet, two errors are to be transformed to warnings.

```
export class TestDataBridge with IModuleTest {
    @IDEBUG(166, "{function(number):void} is not a subtype of {function(T):void}") ①
```

```

@IDEBUG(91, "Incorrect number of arguments: expected 1, got 2.") ②
@Override public run(): void {
    var foo = new Foo(),
        cb = function(val: number): void {},
        db = DataBridge.<number>bind(foo, "bar");
    db.add(cb); ③
    Assert.isTrue(called);
}
}

```

- ① The annotation transforms the error `{function(number):void} is not a subtype of {function(T):void}` into a warning with the following text:

`IDEBUG-166: {function(number):void} is not a subtype of {function(T):void}.`

where `IDEBUG-166` refers to the corresponding bug report, that is `IDEBUG-166`.

- ② This annotation was proposed as a workaround for `IDEBUG-91` which has been fixed.

No error message is produced and an error will be issued on this line instead:

`No matching error found, apparently bug IDEBUG-91 has been fixed or does not occur here.`

- ③ The first error occurs since there is a bug in the IDE type system (as of writing this example) where type arguments are not correctly bound in the case of function expressions used as callback methods.

#### 10.1.5.2. Suppress Warnings



This is not part of the current version

## 10.2. Declaration of Annotations



This is not part of the current version



# Chapter 11. Extended Features

## 11.1. Array and Object Destructuring

N4JS supports array and object destructuring as provided in ES6. This is used to conveniently assign selected elements of an array or object to a number of newly-declared or pre-existing variables or to further destructure them by using nested destructuring patterns<sup>1</sup>.

### 11.1.1. Syntax

```
BindingPattern <Yield>:
    ObjectBindingPattern<Yield>
    | ArrayBindingPattern<Yield>
;

ObjectBindingPattern <Yield> returns BindingPattern:
    {BindingPattern}
    '{' (properties+=BindingProperty<Yield,AllowType=false> (',' properties+=BindingProperty<Yield,AllowType=false>)*)? '}'
;

ArrayBindingPattern <Yield> returns BindingPattern:
    {BindingPattern}
    '['
        elements+=Elision*
        elements+=BindingRestElement<Yield>
        (',' elements+=Elision* elements+=BindingRestElement<Yield>)*
        (',' elements+=Elision*)?
    )?
    ']'
;

BindingProperty <Yield, AllowType>:
    =>(LiteralBindingPropertyName<Yield> ':') value=BindingElement<Yield>
    | value=SingleNameBinding<Yield,AllowType>
;

fragment LiteralBindingPropertyName <Yield>*:
    declaredName=IdentifierName | declaredName=STRING | declaredName=NumericLiteralAsString
    // this is added here due to special treatment for a known set of expressions
    | '[' (declaredName=SymbolLiteralComputedName<Yield> | declaredName=STRING) ']'
;
```

### 11.1.2. Semantics

The following example declares four variables `a`, `b`, `x`, and `prop2`. Variables `a` and `x` will have the value `hello`, whereas `b` and `prop2` will have value 42.

```
var [a,b] = ["hello", 42];
var {prop1:x, prop2} = {prop1:"hello", prop2:42};
```

In the case of `prop2`, we do not provide a property name and variable name separately; this is useful in cases where the property name also makes for a suitable variable name (called `single name binding`).

One of the most useful use cases of destructuring is in a `for...of` loop. Take this example:

```
var arr1 = [ ["hello",1,2,3], ["goodbye",4,5,6] ];
for(var [head,...tail] of arr1) {
    console.log(head,'/',tail);
}
// will print:
// hello / [ 1, 2, 3 ]
// goodbye / [ 4, 5, 6 ]

var arr2 = [ {key:"hello", value:42}, {key:"goodbye", value:43} ];
for(var {key,value} of arr2) {
    console.log(key,'/',value);
```

<sup>1</sup> Further reading on DI Basics: [Fowler04b; Prasanna09a], Verification [Zhu13a; Hudli13a], Frameworks [Lesiecki08a; Betts13a; Knol13a; Dagger]

```
}
// will print:
//   hello / 42
//   goodbye / 43
```

Array and object destructuring pattern can appear in many different places:

- In a variable declaration (not just in variable statements but also in other places where variable declarations are allowed, e.g. plain for loops; called *destructuring binding*; see [Section 9.1.2, “Variable Statement”](#)).
- On the left-hand side of an assignment expression (the assignment expression is then called *destructuring assignment*; see [Section 8.1.21, “Assignment Expression”](#)).
- In a `for..in` or `for..of` loop on the left side of the `in/of` (see [Section 9.2.3, “`for ... of` statement”](#)).



It can also be used in plain statements, but then we actually have one of the above two use cases.

- With lists of formal parameters or function arguments (not supported yet).

For further details on array and object destructuring please refer to the ECMAScript 6 specification - [[ECMA15a](#)].

Type annotations can only be added when a new variable name is introduced since the short version would be ambiguous with the long one. For example:

```
var {x: someTypeOrNewVar} = ol
```

could either mean that a new variable `someTypeOrNewVar` is declared and `ol.x` is assigned to it, or that a new variable `x` is declared with type `someTypeOrNewVar`. The longer form would look like this:

```
var {x: x: someType} = ol
```

We can make this more readable:

```
var {propOfOl: newVar: typeOfNewVar} = ol
```

## 11.2. Dependency Injection

This chapter describes DI mechanisms for N4JS. This includes compiler, validation and language extensions that allow to achieve DI mechanisms built in into the N4JS language and IDE.

N4JS DI support specifies a means for obtaining objects in such a way as to maximize reusability, testability and maintainability, especially compared to traditional approaches such as constructors, factories and service locators. While this can be achieved manually (without tooling support) it is difficult for nontrivial applications. The solutions that DI provides should empower N4JS users to achieve the above goals without the burden of maintaining so-called 'boilerplate' code.

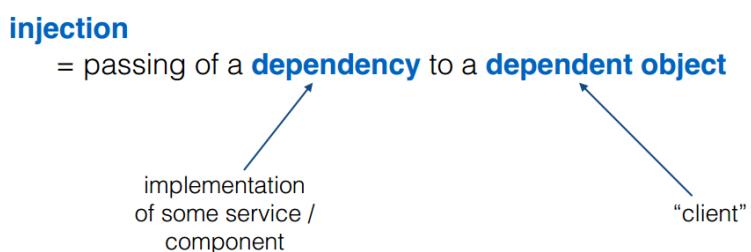


Figure 11.1. DI Basic Terms

key: *pass the dependency instead of letting the client create or find it*

Core terms

- **Service** - A set of APIs describing the functionality of the service.
- **Service Implementations** - One or more implementations of given service API.

- **Client** - Consumer of a given functionality, uses the given **Service Implementation**.
- **Injector** - Object providing **Service Implementation** of a specific **Service**, according to configuration.
- **Binding** - Part of configuration describing which interface implementing a subtype will be injected, when a given interface is requested.
- **Provider** - Factory used to create instances of a given **Service Implementation** or its sub-components, can be a method.
- **Injection Point** - Part of the user's code that will have the given dependency injected. This is usually fields, method parameters, constructor parameters etc.
- **DI configuration** - This describes which elements of the user's code are used in mechanisms and how they are wired. It is derived from user code elements being marked with appropriate annotations, bindings and providers.
- **di wiring** - The code responsible for creating user objects. These are injectors, type factories/providers, fields initiators etc.

### 11.2.1. DI Components and Injectors

N4JS' [Section 11.2, "Dependency Injection"](#) systems is based on the notion of **DIC**.

#### DI Component

**Definition:** [DI Component](#)

A **DIC** is a N4Class annotated with `@GenerateInjector`.

This annotation causes an *injector* to be created for (and associated to) the **DI**. DIC can be composed; meaning that when requested to inject an instance of a type, a DIC's injector can delegate this request to the injector of the containing DIC. An injector always prioritizes its own configuration before delegating to the container's injector. For validation purposes, a child DI can be annotated with `@WithParent` to ensure that it is always used with a proper parent.

*Injector* is the main object of DI mechanisms responsible for creating object graphs of the application. At runtime, injectors are instances of `N4Injector`.

#### DI Component and Injector

**Requirement:** [IDE-138: DI Component and Injector \(ver. 1\)](#)

The following constraints must hold for a class *C* marked as DIC:

1. A subclass *S* of *C* is a DIC as well and it must be marked with `GenerateInjector`.
2. If a parent **DIC** *P* is specified via `WithParent`, then *P* must be a DIC as well.
3. The injector associated to a DIC is of type `N4Injector`. It can be retrieved via `N4Injector.of(DIC)` in which **DIC** is the **DIC**.
4. Injectors associated to DIC a are DI-singletons (cf. [Section 11.2.4.3, "Singleton Scope"](#)). Two calls to `N4Injector.of(DIC)` are different (as different DIC are assumed).

#### Injection Phase

**Requirement:** [IDE-139: Injection Phase \(ver. 1\)](#)

We call the (transitive) creation and setting of values by an injector *I* caused by the creation of a root object *R* the *injection phase*. If an instance *C* is newly created by the injector *I* (regardless of the injection point being used), the injection is transitively applied on *C*. The following constraints have to hold:

1. Root objects are created by one of the following mechanisms:
  - a. Any class or interface can be created as root objects via an injector associated to a DIC:
 

```
var x: X = N4Injector.of(DIC).create(X);
```

 in which **DIC** is a DIC.

Of course, an appropriate binding must exist.<sup>2</sup>

- b. If a type has the injector being injected, e.g. via field injection `@Inject injector: N4Injector;`, then this injector can be used anytime in the control flow to create a new root object similar as above (using `create` method).
  - c. If a provider has been injected (i.e. an instance of `{N4Provider}`), then its `get()` method can be used to create a root object causing a new injection phase to take place.
2. If `C ctor` is marked as injection point, all its arguments are set by the injector. This is also true for an inherited constructor marked as an injection point. See [Req-IDE-143]. For all arguments the injection phase constraints have to hold as well.
  3. All fields of `C`, including *inherited* once, marked as injection points are set by the injector. For all fields the injection phase constraints have to hold as well.

The injector may use a provider method (of a binder) to create nested instances.

The injector is configured with *Binders* and it tracks *Bindings* between types (Section 11.2.2, “Binders and Bindings”). An N4JS developer normally would not interact with this object directly except when defining an entry-point to his application. *Injectors* are configured with *Binders* which contain explicit *Bindings* defined by an N4JS developer. A set of these combined with *implicit bindings* creates the *di configuration* used by a given injector. To configure given *Injectors* with given *Binder(s)* use `@UseBinder` annotation.

### 11.2.1.1. DIComponent Relations

A Parent-Child relation can be established between two DIComponents. Child DIComponents use the parent bindings but can also be configured with their own bindings or *change* targets used by a parent. The final circumstance is local to the child and is referred to as *rebinding*. For more information about bindings see Section 11.2.2, “Binders and Bindings”. A Child-Parent relation is expressed by the `@WithParentInjector` annotation attached to a given DIComponent. When this relation is defined between DIComponents, the user needs to take care to preserve the proper relation between injectors. In other words, the user must provide an instance of the parent injector (the injector of the DIComponent passes as a parameter to `@WithParentInjector`) when creating the child injector (injector of the DIComponent annotated with `@WithParentInjector`).

#### Example 11.1. Simple DIComponents Relation

```
@GenerateInjector
class ParentDIComponent {}

@GenerateInjector
@WithParentInjector(ParentDIComponent)
class ChildDIComponent {}

var parentInjector = N4Injector.of(ParentDiCompoennt);
var childInjector = N4Injector.of(ChildDIComponent, parentInjector);
```

With complex DIComponent structures, injector instances can be created with a directly-declared parent and also with any of its children. This is due to the fact that any child can rebind types, add new bindings, but not remove them. Any child is, therefore, *compatible* with its parents.

#### Compatible DIComponent

**Definition:** Compatible DIComponent

A given DIComponent is compatible with another DIComponent if it has bindings for all keys in other component bindings.

$\exists DIC1, DIC2: DIC1.binding.key \Rightarrow DIC2.binding.key \Rightarrow DIC2 < : DIC1$



Although subtype notation `< :` is used here it does **not** imply actual subtype relations. It was used in this instance for lack of formal notations for DI concepts and because this is similar to the Liskov Substitution principle.

A complex Child-Parent relation between components is depicted in Figure 11.2, “Complex DIComponents Relations” and Example 11.2, “Complex DIComponents Relations” below.

<sup>2</sup> Usually, only the `DIC` itself is created like that, e.g., `var dic = N4Injector.of(DIC).create(DIC);`

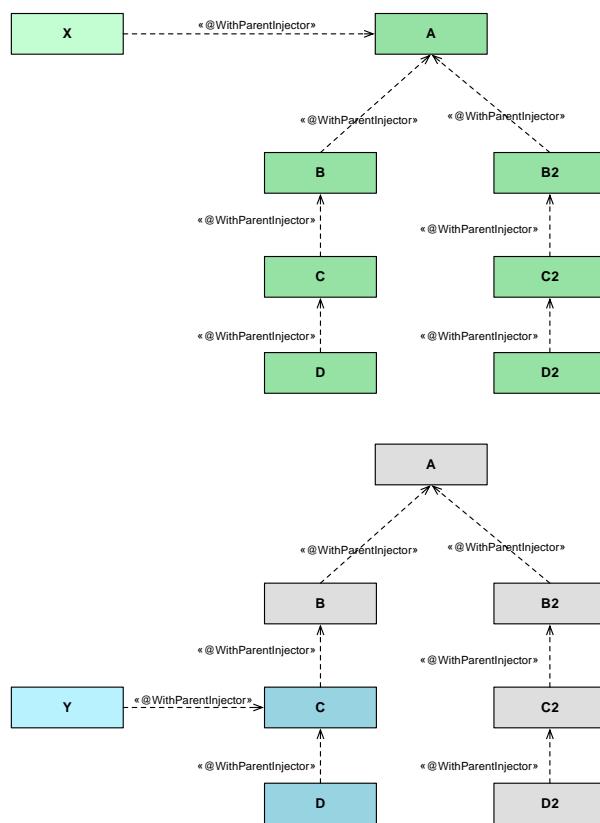


Figure 11.2. Complex DIComponents Relations

**Example 11.2. Complex DIComponents Relations**

```

@GenerateInjector class A {}
@GenerateInjector @WithParentInjector(A) class B {}
@GenerateInjector @WithParentInjector(B) class C {}
@GenerateInjector @WithParentInjector(C) class D {}
@GenerateInjector @WithParentInjector(A) class B2 {}
@GenerateInjector @WithParentInjector(B2) class C2 {}
@GenerateInjector @WithParentInjector(C2) class D2 {}
@GenerateInjector @WithParentInjector(A) class X {}
@GenerateInjector @WithParentInjector(C) class Y {}

// creating injectors
var injectorA = N4Injector.of(A);
//following throws DICConfigurationError, expected parent is not provided
//var injectorB = N4Injector.of(B);
//correct declarations
var injectorB = N4Injector.of(B, injectorA);
var injectorC = N4Injector.of(C, injectorB);
var injectorD = N4Injector.of(D, injectorC);
var injectorB2 = N4Injector.of(B2, injectorA);
var injectorC2 = N4Injector.of(C2, injectorB2);
var injectorD2 = N4Injector.of(D2, injectorC2);

//Any injector of {A,B,C,D,b2,C2,D2} s valid parent for injector of X, e.g. D or D2
N4Injector.of(X, injectorD); //is ok as compatible parent is provided
N4Injector.of(X, injectorD2); //is ok as compatible parent is provided

N4Injector.of(Y, injectorC); //is ok as direct parent is provided
N4Injector.of(Y, injectorD); //is ok as compatible parent is provided

N4Injector.of(Y, injectorB2); //throws DICConfigurationError, incompatible parent is provided
N4Injector.of(Y, injectorC2); //throws DICConfigurationError, incompatible parent is provided
N4Injector.of(Y, injectorD2); //throws DICConfigurationError, incompatible parent is provided

```

**11.2.2. Binders and Bindings**

*Binder* allows an N4JS developer to (explicitly) define a set of *Bindings* that will be used by an *Injector* configured with a given *Binder*. There are two ways for *Binder* to define *Bindings*: `@Bind` (Section 11.2.6.5, “N4JS DI `@Bind`”) annotations and a method annotated with `@Provides`.

*Binder* is declared by annotating a class with the `@Binder` annotation.

A *Binding* is part of a configuration that defines which instance of what type should be injected into an *injection point* (Section 11.2.3, "Injection Points") with an expected type.

*Provider Method* is essentially a *factory method* that is used to create an instance of a type. N4JS allows a developer to declare those methods (see Section 11.2.6.6, "N4JS DI @Provides") which gives them a hook in instance creation process. Those methods will be used when creating instances by the *Injector* configured with the corresponding *Binder*. A provider method is a special kind of binding (*key*) in which the return type of the method is the *key*. The *target* type is unknown at compile time (although it may be inferred by examining the return statements of the provide method).

## Binding

### Definition: Binding

A *binding* is a pair  $\text{bind}(\text{key}, \text{target})$ . It defines that for a dependency with a given key which usually is the expected type at the injection point. An instance of type *target* is injected.

A *binding* is called *explicit* if it is declared in the code, i.e. via `@Bind` annotation or `@Provides` annotation).

A *binding* is called *implicit* if it is not declared. An implicit binding can only be used if the *key* is a class and derived from the type at the injection point, i.e. the type of the field or parameter to be injected. In that case, the *target* equals the *key*.

A provider method *M* (in the binder) defines a binding

$\text{bind}(M.\text{returnType}, X)$

(in which *X* is an existential type with  $\exists X < : \text{target.returnType}$ ).

For simplification, we define:

$$\text{key}^* = \begin{cases} \text{target.returnType}, & \text{if target is provider method} \\ \text{key}, & \text{otherwise (key is a type reference)} \end{cases}$$

and

$$\text{target}^* = \begin{cases} X < : \text{target.returnType}, & \text{if target is provider method} \\ \text{target}, & \text{otherwise (target is a type reference)} \end{cases}$$

## Bindings

### Requirement: IDE-140: Bindings (ver. 1)

For a given binding  $b = (\text{key}, \text{target})$ , the following constraints must hold:<sup>3</sup>

1. *key* must be either a class or an interface.
2. *target* must either be a class or a provider method.
3. If *b* is implicit, then *key* must be a class. If *key* references a type *T*, then *target* = *T* – even if *key* is a use-site structural type.
4. *key* and *target*\* can be nominal, structural or field-structural types, either definition-site or use-site. The injector and binder needs to take the different structural reference into account at runtime!
5. *target*\*  $< : \text{key}$  must hold
6. If during injection phase no binding for a given key is found, an `DIUnsatisfiedBindingError` is thrown.

## Transitive Bindings

### Requirement: IDE-141: Transitive Bindings (ver. 1)

If an injector contains two given bindings  $b_1 = (\text{key}_1, \text{target}_1)$  and  $b_2 = (\text{key}_2, \text{key}_1)$ , an effective binding  $b = (\text{key}_2, \text{target}_1)$  is derived (replacing  $b_1$ ).

N4JS DI mechanisms don't allow for injection of primitives or built-in types. Only user-defined N4Types can be used. In cases where a user needs to inject a primitive or a built-in type, the developer must wrap it into its own class<sup>4</sup>. This is to say that

<sup>3</sup> Note that other frameworks may define other constraints, e.g., arbitrary keys.

<sup>4</sup> Cf. a blog post about micro types - <http://www.markneedham.com/blog/2009/03/10/oo-micro-types/>, and tiny types - <http://darrenhobbs.com/2007/04/11/tiny-types/>

none of the following metatypes can be bound: primitive types, enumerations, functions, object types, union- or intersection types. It is possible to (implicitly) bind to built-in classes.

While direct binding overriding or rebinding is not allowed, *Injector* can be configured in a way where one type can be separately bound to different types with implicit binding, *explicit binding* and in bindings of the child injectors. *Binding precedence* is a mechanism of *Injector* selecting a binding use for a type. It operates in the following order:

1. Try to use explicit binding, if this is not available:
2. Try to delegate to parent injectors (order of lookup is not guaranteed, first found is selected). If this is not available then:
3. Try to use use implicit binding, which is simply to attempt to create the instance.

If no binding for a requested type is available an error will be thrown.

### 11.2.3. Injection Points

By *injection point* we mean a place in the source code which, at runtime, will be expected to hold a reference to a particular type instance.

#### 11.2.3.1. Field Injection

In its simplest form, this is a class field annotated with `@Inject` annotation. At runtime, an instance of the containing class will be expected to hold reference to an instance of the field declared type. Usually that case is called *Field Injection*.

##### Field Injection

**Requirement: IDE-142: Field Injection** (ver. 1)

The injector will inject the following fields:

1. All directly contained fields annotated with `@Inject`.
2. All inherited fields annotated with `@Inject`.
3. The injected fields will be created by the injector and their fields will be injected as well.

##### Example 11.3. Simple Field Injection

Simple Field Injection demonstrates simple field injection using default bindings. Note that all inherited fields (i.e. `A.xInA`) are injected and also fields in injected fields (i.e. `x.y`)

##### Simple Field Injection

```
class X {
    @Inject y: Y;
}
class Y {}

class A {
    @Inject xInA: X;
}
class B extends A {
    @Inject xInB: X;
}

@GenerateInjector
export public class DIC {
    @Inject a: B;
}

var dic = N4Injector.of(DIC).create(DIC);
console.log(dic);           // --> DIC
console.log(dic.a);         // --> B
console.log(dic.a.xInA);    // --> X
console.log(dic.a.xInA.y);  // --> Y
console.log(dic.a.xInB);    // --> X
console.log(dic.a.xInB.y);  // --> Y
```

#### 11.2.3.2. Constructor Injection

Parameters of the constructor can also be injected, in which case this is usually referred to as *Constructor Injection*. This is similar to *Method Injection* and while constructor injection is supported in N4JS, method injection is not (see remarks below).

When a constructor is annotated with `@Inject` annotation, all user-defined, non-generic types given as the parameters will be injected into the instance's constructor created by the dependency injection framework. Currently, optional constructor parameters are always initialized and created by the framework, therefore, they are ensured to be available at the constructor invocation time. Unlike optional parameters, variadic parameters cannot be injected into a type's constructor. In case of annotating a constructor with `@Inject` that has variadic parameters, a validation error will be reported. When a class's constructor is annotated with `@Inject` annotation, it is highly recommended to annotate all explicitly-defined constructors at the subclass level. If this is not done, the injection chain can break and runtime errors might occur due to undefined constructor parameters. In the case of a possible broken injection chain due to missing `@Inject` annotations for any subclasses, a validation warning will be reported.

### Constructor Injection

#### Requirement: IDE-143: Constructor Injection (ver. 1)

If a class *C* has a constructor marked as injection point, the following applies:

1. If *C* is subclassed by *S*, and if *S* has no explicit constructor, then *S* inherits the constructor from *C* and it will be an injection point handled by the injector during injection phase.
2. If *S* provides its own injector, *C.ctor* is no longer recognized by the injector during the injection phase. There will be a warning generated in *S.ctor* to mark it as injection point as well in order to prevent inconsistent injection behavior. Still, *C.ctor* must be called in *S.ctor* similarly to other overridden constructors.

### 11.2.3.3. Method Injection

Other kinds of injector points are method parameters where (usually) all method parameters are injected when the method is called. In a way, constructor injection is a special case of the method itself.

#### 11.2.3.3.1. Provider

`Provider` is essentially a *factory* for a given type. By injecting an `N4Provider` into any injection point, one can acquire new instances of a given type provided by the injected provider. The providers prove useful when one has to solve re-injection issues since the depended type can be wired and injected via the provider rather than the dependency itself and can therefore obtain new instances from it if required. Provider can be also used as a means of delaying the instantiation time of a given type.

`N4Provider` is a public generic built-in interface that is used to support the re-injection. The generic type represents the dependent type that has to be obtained. The `N4Provider` interface has one single public method: `public T get()` which should be invoked from the client code when a new instance of the dependent type is required. Unlike any other unbound interfaces, the `N4Provider` can be injected without any explicit binding.

The following snippet demonstrates the usage of `N4Provider`:

```
class SomeService { }

@singleton
class SomeSingletonService { }

class SomeClass {

    @Inject serviceProvider: N4Provider<SomeService>;
    @Inject singletonServiceProvider: N4Provider<SomeSingletonService>;

    void foo() {
        console.log(serviceProvider.get() ===
            serviceProvider.get()); //false

        console.log(singletonServiceProvider.get() ===
            singletonServiceProvider.get()); //true
    }
}
```

It is important to note that the `N4Provider` interface can be extended by any user-defined interfaces and/or can be implemented by any user-defined classes. For those user-defined providers, consider all binding-related rules; the extended interface, for example, must be explicitly bound via a binder to be injected. The binding can be omitted only for the built-in `N4Provider`s.

### 11.2.4. N4JS DI Life Cycle and Scopes

DI Life Cycle defines when a new instance is created by the injector as its destruction is handled by JavaScript. The creation depends on the scope of the type. Aside from the scopes, note that it is also possible to implement custom scopes and life cycle management via `N4JSProvider` and `Binder@Provides` methods.

### 11.2.4.1. Injection Cycles

#### Injection Cycle

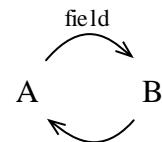
##### Definition: Injection Cycle

We define an injection graph  $G(V, E)$  as a directed graph as follows:  $V$  (the vertices) is the set types of which instances are created during the injection phase and which use.  $E$  (the edges) is a set of directed and labeled edges  $(v_1, v_2, \text{label})$ , where label indicates the injection point:

1.  $(T_O, T_f, \text{"field"})$ , if  $T_f$  is the actualy type of an an injected field of an instance of type  $T_O$
2.  $(T_C, T_p, \text{"ctor"})$ , if  $T_p$  is the type of a parameter used in a constructor injection of type  $T_C$

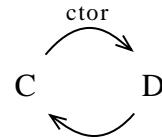
One cycle in this graph is an injection cycle.

When injecting instances into an object, cycles have to be detected and handled independently from the scope. If this is not done, the following examples would result in an infinite loop causing the entire script to freeze until the engine reports an error:



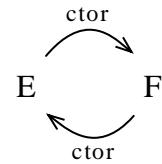
```
class A { @Inject b: B; }
class B { @Inject a: A; }
```

Figure 11.3. Field Cycle



```
class C { @Inject constructor(d: D) {} }
class D { @Inject c: C; }
```

Figure 11.4. Ctor Field Cycle



```
class E { @Inject constructor(f: F) {} }
class F { @Inject e: E; }
```

Figure 11.5. Ctor Cycle

The injector needs to detect these cycles and resolve them.

#### Resolution of Injection Cycles

##### Requirement: IDE-144: Resolution of Injection Cycles (ver. 1)

A cycle  $c \subseteq G$ , with  $G$  being an injection graph, is resolved as follows:

1. If  $c$  contains no edge with  $\text{label} = \text{"ctor"}$ , the cycle is resolved using the algorithm described below.
2. If  $c$  contains at least one edge with  $\text{label} = \text{"ctor"}$ , a runtime exception is thrown.

Cycles stemming from field injection are resolved by halting the creation of new instances of types which have been already created by a containing instance. The previously-created instance is then reused. This makes injecting the instance of a (transitive) container less complicated and without the need to pass the container instance down the entire chain. The following pseudo code describes the algorithm to create new instances which are injected into a newly created object:

```
function injectDependencies(object) {
    doInjectionWithCycleAwareness(object, {typeof object -> object})
}
```

```

function doInjectionWithCycleAwareness(object, createdInstancesPerType) {
    forall v $\in$ injectedVars of object {
        var type = retrieveBoundType(v)
        var instance = createdInstancesPerType.get(type)
        if (not exists instance) {
            instance = createInstance(type, createdInstancesPerType)
            doInjectionWithCycleAwareness(instance,
                createdInstancesPerType $\cap$ ${\{(type->instance)\}})
        }
        v.value = instance;
    }
}

```

The actual instance is created in line 10 via `createInstance`. This function then takes scopes into account. The `createdInstancesPerType` map is passed to that function in order to enable cycle detection for constructor injection. The following scopes are supported by the N4JS DI, other scopes, cf. [Jersey custom scopes](#) and [Guice custom scopes](#), may be added in the future.

This algorithm is not working for constructor injection because it is possible to already access all fields of the arguments passed to the constructor. In the algorithm, however, the instances may not be completely initialized.

#### 11.2.4.2. Default Scope

The default scope always creates a new instance.

#### 11.2.4.3. Singleton Scope

The singleton scope (per injector) creates one instance (of the type with `@Singleton` scope) per injector, which is then shared between clients.

The injector will preserve a single instance of the type of `s` and will provide it to all injection points where type of `s` is used. Assuming nested injectors without any declared binding where the second parameter is `s`, the same preserved singleton instance will be available for all nested injectors at all injection points as well.

The singleton preservation behavior changes when explicit bindings are declared for type `s` on the nested injector level. Let's assume that the type `s` exists and the type is annotated with `@Singleton`. Furthermore, there is a declared binding where the binding's second argument is `s`. In that case, unlike in other dependency injection frameworks, nested injectors may preserve a singleton for itself and all descendant injectors with `@Bind` annotation. In this case, the preserved singleton at the child injector level will be a different instance than the one at the parent injectors.

The tables below depict the expected runtime behavior of singletons used at different injector levels. Assume the following are injectors: `C`, `D`, `E`, `F` and `G`. Injector `C` is the top most injector and its nesting injector `D`, hence injector `C` is the parent of the injector `D`. Injector `D` is nesting `E` and so on. The most nested injector is `G`. Let's assume `J` is an interface, class `U` implements interface `J` and class `V` extends class `U`. Finally assume both `U` and `V` are annotated with `@Singleton` at definition-site.

[Table 11.1, “DI No Bindings”](#) depicts the singleton preservation for nested injectors without any bindings. All injectors use the same instance from a type. Type `J` is not available at all since it is not bound to any concrete implementation:

**Table 11.1. DI No Bindings**

Binding					
Injector nesting (>)	C	D	E	F	G
<code>J</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>	<code>NaN</code>
<code>U</code>	<code>U<sub>0</sub></code>	<code>U<sub>0</sub></code>	<code>U<sub>0</sub></code>	<code>U<sub>0</sub></code>	<code>U<sub>0</sub></code>
<code>V</code>	<code>V<sub>0</sub></code>	<code>V<sub>0</sub></code>	<code>V<sub>0</sub></code>	<code>V<sub>0</sub></code>	<code>V<sub>0</sub></code>

[Table 11.2, “DI Transitive Bindings”](#) is configured by explicit bindings. At the root injector level, type `J` is bound to type `U`. Since the second argument of the binding is declared as a singleton at the definition-site, this explicit binding implicitly ensures that the injector and all of its descendants preserve a singleton of the bound type `U`. At injector level `C`, `D` and `E`, the same instance is used for type `J` which is type `U` at runtime. At injector level `E` there is an additional binding from type `U` to type `V` that overrules the binding declared at the root injector level. With this binding, each places where `J` is declared, type `U` is used at runtime.

Furthermore, since `V` is declared as a singleton, both injector `F` and `G` are using a shared singleton instance of type `V`. Finally, for type `V`, injector `C`, `D` and `E` should use a separate instance of `V` other than injector level `F` and `G` because `V` is preserved at injector level `F` with the `U → V` binding.

**Table 11.2. DI Transitive Bindings**

Binding	$J \rightarrow U$			$U \rightarrow V$	
<b>Injector nesting (&gt;)</b>	C	D	E	F	G
<b>J</b>	$U_0$	$U_0$	$U_0$	$V_0$	$V_0$
<b>U</b>	$U_0$	$U_0$	$U_0$	$V_0$	$V_0$
<b>V</b>	$V_1$	$V_1$	$V_1$	$V_0$	$V_0$

Table 11.3, "DI Re - Binding" depicts the singleton behaviour but unlike the above table, the bindings are declared for the interface **J**.

**Table 11.3. DI Re - Binding**

Binding	$J \rightarrow U$			$J \rightarrow V$	
<b>Injector nesting (&gt;)</b>	C	D	E	F	G
<b>J</b>	$U_0$	$U_0$	$U_0$	$V_0$	$V_0$
<b>U</b>	$U_0$	$U_0$	$U_0$	$U_0$	$U_0$
<b>V</b>	$V_1$	$V_1$	$V_1$	$V_0$	$V_0$

Table 11.4, "DI Child Binding" describes the singleton behavior when both bindings are configured at child injector levels but not the root injector level.

**Table 11.4. DI Child Binding**

Binding		$U \rightarrow V$		$J \rightarrow U$	
<b>Injector nesting (&gt;)</b>	C	D	E	F	G
<b>J</b>	$NaN$	$NaN$	$NaN$	$U_0$	$U_0$
<b>U</b>	$U_1$	$V_0$	$V_0$	$U_0$	$U_0$
<b>V</b>	$V_1$	$V_0$	$V_0$	$V_0$	$V_0$

#### 11.2.4.4. Per Injection Chain Singleton

The per injection chain singleton is 'between' the default and singleton scope. It can be used in order to explicitly describe the situation which happens when a simple cycle is resolved automatically. It has more effects that lead to a more deterministic behavior.

Assume a provider declared as

```
var pb: Provider<B>;
```

to be available:

```
@PerInjectionSingleton
class A { }

class B { @Inject a: A; @Inject a1: A; }

b1=pb.get();
b2=pb.get();
b1.a != b2.a
b1.a == b1.a1
b2.a == b2.a1
```

```
@Singleton
class A { }

class B { @Inject a: A; @Inject a1: A; }

b1=pb.get();
b2=pb.get();
b1.a == b2.a
b1.a == b1.a1
b2.a == b2.a1
```

```
// no annotation
class A { }
```

```
class B { @Inject a A; @Inject al: A; }

b1=pb.get();
b2=pb.get();
b1.a != b2.a
b1.a != b1.al
b2.a != b2.al
```

## 11.2.5. Validation of callsites targeting N4Injector methods

Terminology for this section:

- a value is **injectable** if it
  - either conforms to a user-defined class or interface (a non-parameterized one, that is),
  - or conforms to Provider-of-T where T is injectable itself.
- a classifier declaring injected members is said to **require injection**

To better understand the validations in effect for callsites targeting

```
N4Injector.of(ctorOfDIC: constructor{N4Object}, parentDIC: N4Injector?, ...providedBinders: N4Object)
```

we can recap that at runtime:

- The first argument denotes a DIC constructor.
- The second (optional) argument is an injector.
- Lastly, the purpose of **providedBinders** is as follows:
  - The DIC above is marked with one or more `@UseBinder`.
  - Some of those binders may require injection.
  - Some of those binders may have constructor(s) taking parameters.
  - The set of binders described above should match the providedBinders.

Validations in effect for `N4Injector.create(type{T} ctor)` callsites:

- `type{T}` should be injectable (in particular, it may be an `N4Provider`).

## 11.2.6. N4JS DI Annotations

Following annotations describe API used to configure N4JSDI.

### 11.2.6.1. N4JS DI `@GenerateInjector`

<b>name</b> <code>@GenerateInjector</code>	<b>targets</b> N4Class	<b>retention policy</b>	RUNTIME
<b>transitive</b> NO	<b>repeatable</b> NO	<b>arguments</b>	NO

`@GenerateInjector` marks a given class as DIComponent of the graph. The generated injector will be responsible for creating an instance of that class and all of its dependencies.

### 11.2.6.2. N4JS DI `@WithParentInjector`

<b>name</b> <code>@WithParentInjector</code>	<b>targets</b> N4Class	<b>retention policy</b>	RUNTIME
<b>transitive</b> NO	<b>repeatable</b> NO	<b>arguments</b>	TypeRef

`@WithParentInjector` marks given *injector* as depended on other *injector*. The depended *injector* may use provided *injector* to create instances of objects required in its object graph.

Additional *WithParentInjector* constraints:

DI WithParentInjector

**Requirement: IDE-145:** DI WithParentInjector (ver. 1)

1. Allowed only on `N4ClassDeclarations` annotated with `@GenerateInjector`.
2. Its parameter can only be `N4ClassDeclarations` annotated with .

### 11.2.6.3. N4JS DI @UseBinder

<b>name</b>	@UseBinder	<b>targets</b>	N4Class	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>arguments</b>	TypeRef	<b>arguments are optional</b>	NO

`@UseBinder` describes *Binder* to be used (configure) target *Injector*.

DI UseInjector

**Requirement: IDE-146: DI UseInjector** (ver. 1)

1. Allowed only on `N4ClassDeclarations` annotated with `@GenerateInjector`. 2. Its parameter can only be `N4ClassDeclarations` annotated with `@Binder`.

### 11.2.6.4. N4JS DI @Binder

<b>name</b>	@Binder	<b>targets</b>	N4Class	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>repeatable</b>	NO	<b>arguments</b>	NONE

`@Binder` defines a list of bind configurations. That can be either `@Bind` annotations on `@Binder` itself or its factory methods annotated with `@Provides`.

DI binder

**Requirement: IDE-147: DI binder** (ver. 1)

1. Target `N4ClassDeclaration` must not be *abstract*. 2. Target `N4ClassDeclaration` must not be annotated with `@GenerateInjector`. 3. Target class cannot have *injection points*.

### 11.2.6.5. N4JS DI @Bind

<b>name</b>	@Bind	<b>targets</b>	N4ClassDeclaration	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>arguments</b>	TypeRef key, TypeRef target	<b>arguments are optional</b>	NO

Defines *binding* between type and subtype that will be used by injector when configured with target [Section 11.2.6.4, “N4JS DI @Binder”](#). See also [Section 11.2.5, “Validation of callsites targeting N4Injector methods”](#) for description of injectable types.

DI Bind

**Requirement: IDE-148: DI Bind** (ver. 1)

1. Allowed only on `N4ClassDeclarations` that are annotated with `@Binder` ([Section 11.2.6.4, “N4JS DI @Binder”](#)). 2. Parameters are instances of one of the values described in [Section 11.2.5, “Validation of callsites targeting N4Injector methods”](#). 3. The second parameter must be a subtype of the first one.

### 11.2.6.6. N4JS DI @Provides

<b>name</b>	@Provides	<b>targets</b>	N4MethodDeclaration	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>repeatable</b>	NO	<b>arguments</b>	NONE

`@Provides` marks *factory method* to be used as part `DI`. This is treated as *explicit binding* between declared return type and actual return type. This method is expected to be part of the `@Binder`. Can be used to implement custom scopes.

DI Provides

**Requirement: IDE-149: DI Provides** (ver. 1)

1. Allowed only on `N4MethodDeclarations` that are part of a classifier annotated with `@Binder`.
2. Annotated method declared type returns instance of one of the types described in *injectable values* [Section 11.2.5, “Validation of callsites targeting N4Injector methods”](#).

### 11.2.6.7. N4JS DI @Inject

<b>name</b>	<code>@Inject</code>	<b>targets</b>	N4Field, N4Method, constructor	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>repeatable</b>	NO	<b>arguments</b>	NO

`@Inject` defines the injection point into which an instance object will be injected. The specific instance depends on the injector configuration (bindings) used. Class fields, methods and constructors can be annotated. See Section 11.2.3, “Injection Points” for more information.

#### DI Inject

##### Requirement: IDE-150: DI Inject (ver. 1)

1. Injection point bindings need to be resolvable.
2. Binding for given type must not be duplicated.
3. Annotated types must be instances of one of the types described in Section 11.2.5, “Validation of callsites targeting N4Injector methods”.

### 11.2.6.8. N4JS DI @Singleton

<b>name</b>	<code>@Singleton</code>	<b>targets</b>	N4Class	<b>retention policy</b>	RUNTIME
<b>transitive</b>	NO	<b>repeatable</b>	NO	<b>arguments</b>	NO

In the case of annotating a class `s` with `@Singleton` on the definition-site, the singleton scope will be used as described in Section 11.2.4.3, “Singleton Scope”.

## 11.3. Test Support

N4JS provides some annotations for testing. Most of these annotations are similar to annotations found in JUnit 4. For details see our Mangelhaft test framework (stdlib specification) and the N4JS-IDE specification.

In order to enable tests for private methods, test projects may define which project they are testing.

#### Test API methods and types

##### Requirement: IDE-151: Test API methods and types (ver. 1)

In some cases, types or methods are only provided for testing purposes. In order to improve usability, e.g. content assist, these types and methods can be annotated with `@TestAPI`. There are no constraints defined for that annotation at the moment.

## 11.4. Polyfill Definitions

In plain JavaScript, so called *polyfill* (or sometimes called *shim*) libraries are provided in order to modify existing classes which are only prototypes in plain JavaScript. In N4JS, this can be defined for declarations via the annotation `@Polyfill` or `@StaticPolyfill`. One of these annotations can be added to class declarations which do not look that much different from normal classes. In the case of polyfill classes, the extended class is modified (or filled) instead of being subclassed.

We distinguish two flavours of polyfill classes: runtime and static.

- Runtime polyfilling covers type enrichment for runtime libraries. For type modifications the annotation `@Polyfill` is used.
- Static polyfilling covers code modifications for adapting generated code. The annotation `@StaticPolyfill` denotes a polyfill in ordinary code, which usually provides executable implementations.

#### Polyfill Class

##### Definition: Polyfill Class

A *polyfill class* (or simply *polyfill*) is a class modifying an existing one. The polyfill is not a new class (or type) on its own. Instead, new members defined in the polyfill are added to the modified class and existing members can be modified similarly to overriding. We call the modified class the *filled class* and the modification *filling*.

We add a new pseudo property *polyfill* to classes in order to distinguish between normal (sub-) classes and polyfill classes.

### Polyfill Class

**Requirement: IDE-152: Polyfill Class (ver. 1)**

For a polyfill class  $P$  annotated with `@Polyfill` or `@staticPolyfill`, that is  $P.polyfill = \text{true}$ , all the following constraints must hold:

1.  $P$  must extend a class  $F$ ,  $F$  is called the filled class:

$$P.super = F$$

2.  $P$ 's name equals the name of the filled class and is contained in a module with same qualified name (specifier or global):

$$\begin{aligned} P.name &= F.name \\ \wedge P.containedModule.global &= F.containedModule.global \\ \wedge (P.containedModule.global \\ \vee P.containedModule.specifier &= F.containedModule.specifier) \end{aligned}$$

3. Both the polyfill and filled class must be top-level declarations (i.e., no class expression):

$$P.topLevel = \text{true} \wedge F.topLevel = \text{true}$$

4.  $P$  must not implement any interfaces:

$$P.implementedInterfaces = \emptyset$$

5.  $P$  must have the same access modifier (access, abstract, final) as the filled class:

$$\begin{aligned} P.accessModifier &= F.accessModifier \\ P.abstract &= F.abstract \\ P.final &= F.final \end{aligned}$$

6. If  $P$  declares a constructor, it must be override compatible with the constructor of the filled class:

$$\exists P.ownedCtor : P.ownedCtor < : F.ctor$$

7.  $P$  must define the same type variables as the filled class  $F$  and the arguments must be in the same order as the parameters (with no further modifications):

$$\begin{aligned} \forall i, 0 \leq i < |P.typeParams| : \\ P.typeParams_i &= F.typeParams_i \\ \wedge P.typeParams_i.name &= P.super.typeArgs_i.name \end{aligned}$$

8. All constraints related to member redefinition (cf. [Section 5.2.8, “Redefinition of Members”](#)) have to hold. In the case of polyfills, this is true for constructors (cf. [\[Req-IDE-72\]](#)) and private members.

#### 11.4.1. Runtime Polyfill Definitions

(Runtime) Libraries often do not provide completely new types but modify existing types. The ECMA-402 Internationalization Standard [\[ECMA12a\]](#), for example, changes methods of the built-in class `Date` to be timezone aware. Other scenarios include new functionality provided by browsers which are not part of an official standard yet. Even ECMAScript 6 [\[ECMA15a\]](#) extends the predecessor [\[ECMA11a\]](#) in terms of new methods (or new method parameters) added to existing types (it also adds completely new classes and features, of course).

Runtime polyfills are only applicable to runtime libraries or environments and thus are limited to n4jsd files.

### Runtime Polyfill Class

**Requirement: IDE-153: Runtime Polyfill Class (ver. 1)**

For a runtime-polyfill class  $P$  annotated with `@Polyfill`, that is  $P.staticpolyfill = \text{false}$ , all the following constraints must hold in addition to [\[Req-IDE-152\]](#):

1. Both the polyfill and filled class are provided by the runtime (annotated with `@ProvidedByRuntime`): <sup>5</sup>

$$P.providedByRuntime = \text{true} \wedge F.providedByRuntime = \text{true}$$

<sup>5</sup> This restriction has two reasons: Firstly, user-defined types with implementations would require to ‘bootstrap’ the polyfill, which is impossible to do automatically without serious constraints on bootstrap code in general. Secondly, instead of filling user-defined types, they can be subclasses. Mechanisms such as dependency injection could then solve almost all remaining problems.

## Applying Polyfills

### Requirement: IDE-154: Applying Polyfills (ver. 1)

A polyfill is automatically applied if a runtime library or environment required by the current project provides it. In this case, the following constraints must hold:

1. No member must be filled by more than one polyfill.

## 11.4.2. Static Polyfill Definitions

Static polyfilling is a compile time feature to enrich the definition and usually also the implementation of generated code in N4JS. It is related to runtime polyfilling described in [Section 11.4.1, “Runtime Polyfill Definitions”](#) in a sense that both fillings enrich the types they address. Despite this, static polyfilling and runtime polyfilling differ in the way they are handled.

Static polyfills usually provide executable implementations and are thus usually found in n4js files. However, they are allowed in n4jsd files, as well, for example to enrich generated code in an API project.

The motivation for static polyfills is to support automatic code generation. In many cases, automatically generated code is missing some information to make it sufficiently usable in the desired environment. Manual enhancements usually need to be applied.

If we think of a toolchain, the question may arise how to preserve the manual work when a regeneration is triggered. Static polyfilling allows the separation of generated code and manual adjustments in separate files. The transpiler merges the two files into a single transpiled file. To enable this behaviour, the statically fillable types must be contained in a module annotated with `@StaticPolyfillAware`. The filling types must also be annotated with `@StaticPolyfill` and be contained in a different module with same specifier but annotated with `@StaticPolyfillModule`. Static polyfilling is restricted to a project, thus the module to be filled as well as the filling module must be contained in the same project.

We add a new pseudo property `staticPolyfill` to classes in order to distinguish between normal (sub-) classes and static polyfill classes. We add two new pseudo properties to modules in order to modify the transpilation process. The mutually-exclusive properties `staticPolyfillAware` and `staticPolyfill` signal the way these files are processed.

In order to support efficient transpilation, the following constraint must hold in addition to constraints:

## Static Polyfill Layout

### Requirement: IDE-155: Static Polyfill Layout (ver. 1)

For a static polyfill class  $P$  annotated with `@StaticPolyfill`, that is  $P.staticpolyfill = \text{true}$ , all the following constraints must hold in addition to [Req-IDE-152]:

1.  $P$ 's name equals the name of the filled class and is contained in a module with the same qualified name:

$$\begin{aligned} P.name &= F.name \\ \wedge P.containedModulespecifier &= F.containedModulespecifier \end{aligned}$$

2. Both the static polyfill and the filled class are part of the same project:

$$P.project = F.project$$

3. The filled class must be contained in a module annotated with `@StaticPolyfillAware`:

$$F.containedModule.staticPolyfillAware = \text{true}$$

4. The static polyfill and the filled type must both be declared in an n4js file or both in an n4jsd file.

5. The filling class must be contained in a module annotated with `@StaticPolyfillModule`:

$$P.containedModule.staticPolyfillModule = \text{true}$$

6. For a statically-filled class  $F$  there is at most one static polyfill:

$$(P_1 \text{ is static polyfill of } F \wedge P_2 \text{ is static polyfill of } F) \rightarrow P_1 = P_2$$

## Restrictions on static polyfilling

### Requirement: IDE-156: Restrictions on static polyfilling (ver. 1)

For a static polyfilling module  $M_P$  the following must hold:

1. All top-level elements are static polyfills:

$$\frac{\forall T \in M_P \wedge T.\text{topLevel} = \text{true}}{T.\text{staticPolyfill} = \text{true}}$$

2. It exists exactly one filled module  $M_F$  annotated with  $\text{staticPolyfillAware}$  in the same project.

3. It is an error if two static polyfill modules for the same filled module exist in the same project:

$$\frac{M_1.\text{specifier} = M_2.\text{specifier} \wedge M_1.\text{project} = M_2.\text{project} \wedge M_1.\text{staticPolyfillModul} = M_2.\text{staticPolyfillModul} = \text{true}}{M_1 = M_2}$$

#### Example 11.4. Static polyfill

[Static Polyfill, Genmod](#) shows an example of generated code. [Static Polyfill, Polyfillmod](#) demonstrates the static polyfill.

Note that the containing project has two source folders configured:

`Project/src/n4js` and `Project/src/n4jsgen`.

#### Static Polyfill, Polyfillmod

```
@@StaticPolyfillAware
export public class A {
    constructor() {...}
    m1(): void{...}
}
export public class B {
    constructor() {...}
    m2(): void{...}
}
```

#### Static Polyfill, Genmod

```
@@StaticPolyfillModule
@StaticPolyfill
export public class B extends B {
    @Override
    constructor(){ ... } // replaces generated ctor of B
    @Override
    m1(): void {...} // adds overridden method m1 to B
    @Override
    m2(): void {...} // replaces method m2 in B
    m3(): void {...} // adds new method m3 to B
}
```

### 11.4.3. Transpiling static polyfilled classes

Transpiling static polyfilled classes encounters the special case that two different `n4js` source files with the same qualified name are part of the project. Since the current transpiler is file-based, both files would be transpiled to the same output destination and would therefore overwrite each other. The following pre-transpilation steps handle this situation:

- Current file to transpile is  $M$
- If  $M.\text{staticPolyfillAware} = \text{true}$ , then
  - search for a second file  $G$  with same qualified name:  
 $G.\text{specifier} = M.\text{specifier} \wedge G.\text{project} = M.\text{project}$
  - If  $\exists G$ , then
    - merge  $G$  into current file  $M \rightarrow M'$
    - conventionally transpile  $M'$
    - else conventionally transpile  $M$
- else, if  $M.\text{staticPolyfillModule} = \text{true}$ ,
  - then do nothing. (Transpilation will be triggered for filled type separately.)
- else, conventionally transpile  $M$



# Chapter 12. Components

## 12.1. N4JS Platform Architecture

### 12.1.1. Overview

Figure 12.1, “N4JS Component Overview” shows the N4JS components described in detail in this chapter.<sup>1</sup>

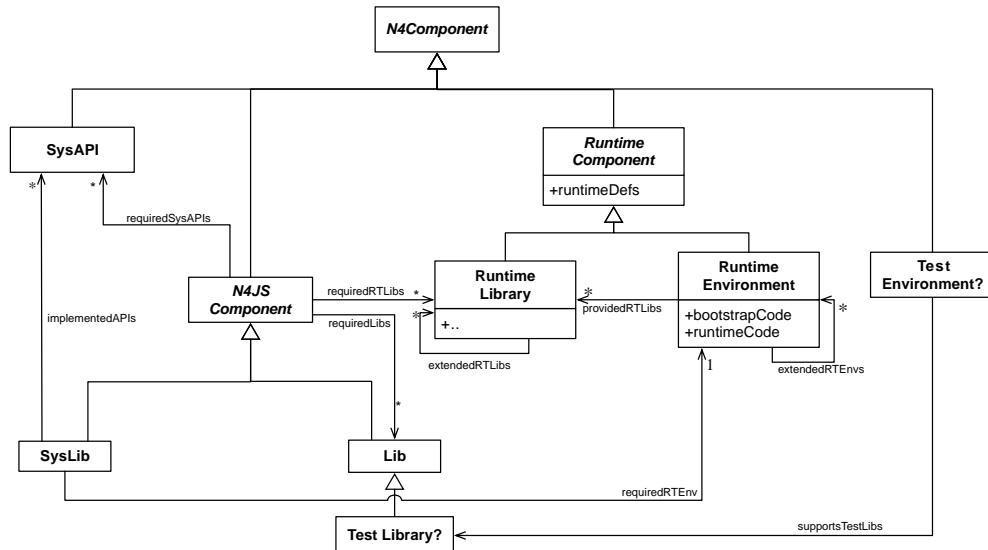


Figure 12.1. N4JS Component Overview

The N4JS platform distinguishes several types of components. The following components can only be created by internal developers:

<b>Runtime Environment</b>	Definition of a runtime environment such as ECMAScript 5. A Runtime Environment describes the base types provided by the runtime directly which are usually globally available. Other components do not directly rely on runtime environments, but on runtime libraries.
<b>Runtime Library</b>	Collections of types provided by the runtime. These types may be extensions to certain language specifications. E.g., the ECMAScript 6 collection classes are already provided by some environments otherwise only supporting ECMAScript 5. The collections are defined in terms of a runtime library which can then be provided by these environments. Runtime libraries may also contain polyfills to alter types predefined in the environment.
<b>Test Environment</b>	Not yet clear. Environments defined for tests.
<b>Test Library</b>	Not yet clear. Libraries defined for tests supported by the to enable running tests and viewing test reports directly within the , such as <i>Mangelhaft</i> .

The following components can be created by external developers:

<b>Apps</b>	User-written N4JS applications running in a web browser (the reference browser is Chrome).
<b>Processors</b>	User-written N4JS processors running on Node.js.
<b>Libraries</b>	User-written libraries used by apps, processors or other libraries.

These components are described in detail later.

A component is similar to a single project in the N4JS IDE. Generally, it contains the following:

<b>Package.json File</b>	The <code>package.json</code> file describing the components, dependencies and metadata.
<b>Resources</b>	Resources such as images, N4ML files etc.
<b>Sources</b>	Source files of modules - actual N4JS files used in a project.
<b>Compilation</b>	Compiled, minified and concatenated versions of the N4JS files and other JS files.
<b>Tests</b>	Optional test sources and compiled tests.
<b>Source Maps</b>	Optional source maps.

<sup>1</sup> Note that this diagram does not necessarily reflect the actual internal implementation but only the external view.

Components contain modules. Figure 12.2, “Content of a Component” describes what can be contained in a component.

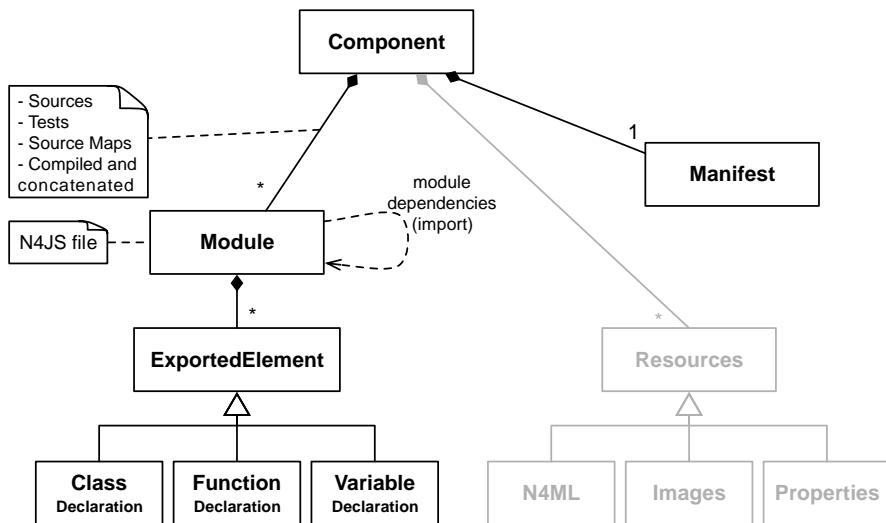


Figure 12.2. Content of a Component

## 12.2. Component Types

Different N4JS component types are described in this section. At compile time and runtime, dependent components have to be available. This is the responsibility of the user interface ([IDE](#) or [CLI](#)) and described in the [N4IDE](#) specification.

### 12.2.1. Libraries

A library is a user project providing modules with declaration.

### 12.2.2. Runtime Environment and Runtime Libraries

Runtime environments and libraries define globally available elements (types, variables, functions) provided by the JavaScript engine. Both must contain *only* definition files (n4jsd) of which all elements are marked as `@ProvidedByRuntime` (Section 13.4, “Runtime Definitions”) and `@Global` (Section 13.3, “Global Definitions”).

Other projects may refer to *multiple* runtime libraries in their `package.json` file via the property `requiredRuntimeLibraries`.

The concrete runtime environment and library are selected by the JavaScript engine. Deployment and execution scripts must ensure that a component can run on the given engine; the required environments and libraries must all be compatible with the provided environment. If no runtime environment is specified, a default an ECMAScript 5 runtime is assumed to be present.

Typical runtime environments are ES5 or ES6, typical runtime libraries are DOM or HTML.

In JavaScript, browsers and other execution environments provide built-in objects. In browsers, for example, the whole DOM is made available via built-in object types. In this case, even the global object also becomes a different type (in N4JS terms). Besides execution environments such as browsers or Node.js, libraries also provide functionality by exposing globally available functions. This is often used to bridge execution environment inconsistencies. When browser API differences are adapted by a library, this is called a *polyfil*. Other adaptations, such as enabling ECMAScript 6 object types in ECMAScript 5 environments, are known as *shim*. Instead of directly supporting these kind of ‘hacks’, other components specify which runtime environment and libraries they depend on by specifying unique runtime ids. Possible shims (in case of environments) or polyfills (in case of libraries) are transparently provided by the execution environment and the bootstrap code.

### 12.2.3. Tests

Tests are special projects which contain tests for other projects.

#### Test Project

**Requirement:** [IDE-157: Test Project \(ver. 1\)](#)

1. Tests have full access to the tested project including elements with `project` visibility.
2. Only other test projects can depend on tests project. In other words, other components must not depend on test components.

In a test project, the tested projects can be specified via `testee`.

## 12.2.4. Type Definitions

Projects of type "definition" are special projects which only provide type information for another so-called *implementation project*, which only provides executable JS files.

Generally, client projects that depend on a given *implementation project* may additionally declare a dependency on a corresponding type definitions project, in order to integrate type information on the implementation project. This is implemented by means of module-level shadowing. More specifically, given a client imports a module with module specifier  $M$  from the implementation project. When resolving the module specifier,  $M$  will first be resolved against the implementation project's type definitions and only secondarily against the implementation project. As a consequence, type definition projects may only provide partial type information, while the remaining modules of the implementation project remain accessible through dynamic namespace imports.

### Type Definition Project Configuration

**Requirement:** GH-821002: [Type Definition Project Configuration](#) (ver. 1)

For type definition projects, the following constraints must hold true with regard to their project configuration:

1. They must declare their *implementation project* via the `definesPackage` property in their `package.json` file.
2. They must not declare an output folder.

## 12.3. Component Content

A component is similar to a project in the N4JS IDE. It consists of sources and test sources. These items are contained in separate folders alongside output folders and settings specified in the `package.json` file. The `package.json` file serves as the project description file and is stored at the root of the project (see [Package.json File](#) for details).

For build and production purposes, other files such as `pom.xml` or `.project` files are automatically derived from the `package.json`. These files are not to be added manually.

## 12.4. Package.json File

A folder is a "component" if and only if it contains a `package.json` file. Being a component means that this folder is recognized by all N4JS-related tools but does not necessarily mean the component contains N4JS code (it could just contain plain Javascript). The main benefit of being a component in this sense is that this unit of code can be used from N4JS components as a dependency.

For example, a plain npm project containing only plain Javascript can be a component and can therefore be used as a project dependency of a full-blown N4JS project.

### 12.4.1. Basic Properties

The following standard `package.json` properties are used by N4JS tooling. Unless otherwise noted, all these properties have the exact same format and meaning as usual in `package.json` files.

**name**      Used as the globally unique identifier of the component.

**version**     The component's version.

**dependencies**   List of components required at runtime and compile time.

**devDependencies**   List of components required at compile time only.

**main**       Path relative to the component's root folder, pointing to a `.js` file located in a source container (the `.js` file extension is optional, i.e. may be omitted). This file then serves as the component's default entry point, i.e. project imports pointing to this component from other components will import from the file denoted by this property. In addition, this property may denote a folder and is then assumed to point to a file `index.js` located in that folder. If this property denotes a file other than a `.js` file, it will be ignored. In particular, it cannot be used for `.n4js` files; in that case, property "mainModule" has to be used (see below).

### 12.4.2. N4JS Properties

In addition to the standard properties above, there is a single N4JS-specific top-level property called "n4js". The value of this property must always be an object that may have any combination of the following properties:

**projectType**    (string) Must be one of the following strings:

**application**      An application. See [??TITLE???](#).

**library**          A library. See [Section 12.2.1, "Libraries"](#).

<b>processor</b>	For processors running server-side on the N4 platform. Not implemented yet.
<b>test</b>	An N4JS project containing tests for one or more other N4JS projects specified via property "testedProjects".
<b>api</b>	For N4JS projects that contain only API (in <code>.n4jsd</code> files) to be implemented by other, so-called implementation projects. See properties "implementationId", "implementedProjects".
<b>runtimeEnvironment</b>	Runtime environments. See <a href="#">Section 12.6.1, “Runtime Environment Resolution”</a> .
<b>runtimeLibrary</b>	Runtime libraries. See <a href="#">Section 12.2.2, “Runtime Environment and Runtime Libraries”</a> .
<b>validation</b>	A project in which <code>.n4js</code> files are only being validated, not transpiled. This is used for projects that are implemented in terms of <code>.js</code> files but that also provide type information in terms of <code>.n4jsd</code> files.
<b>plainjs</b>	A project which only contains <code>.js</code> files and no N4JS resources. The contained JS files are only indexed to allow for dynamic imports of specific JavaScript modules. Projects of this type are not being transpiled.
<b>vendorId</b>	(string) Globally unique identifier for the component's vendor. Used for the <code>@Internal</code> accessibility modifier.
<b>vendorName</b>	(string) Human-readable name of the component's vendor. Used only for informational purposes.
<b>output</b>	(string) Path relative to the component's root folder, pointing to a folder where all output files will be placed. In particular, this is where the N4JS transpiler will put the <code>.js</code> files created for each <code>.n4js</code> file.
<b>sources</b>	(object) Defines various sub-folders where sources, etc. are located. All properties of the given object must have to following format: the name must be "source", "external", or "test"; the value must be an array of strings, with each string defining a path relative to the component's root folder, pointing to a folder where source files of the corresponding type are located. For example, paths given via name "source" tell the N4JS transpiler where to look for <code>.n4js</code> source files to be compiled.
<b>moduleFilters</b>	(object) Filters for fine-tuning the validator and compiler. A filter is applied to modules matching the given module specifier which may contain wildcards, optionally restricted to modules defined in a specific source path.  All properties of the given object must have the following format: the name must be a valid module filter type (see below); the value must be an array of strings, with each string defining a pattern of files inside one of the source containers for which validation or module wrapping is to be turned off. Instead of a plain string, the inner array may contain an object with properties "module" and "sourceContainer" to make this filter apply to only one of the source containers (instead of all source containers, which is the default).  <b>noValidate</b> Modules matching this filter are not semantically validated. That is, they are still syntactically validated. If they are contained in source or test source fragments, it must be possible to bind references to declarations inside these modules. Note that switching off validation for n4js files is disallowed. <b>noModuleWrap</b> Files matching this filter are not wrapped into modules and they are not semantically validated. Since they are assumed to be wrapped into modules, declarations inside these modules cannot be referenced by n4js code.

**Example 12.1. Module Filters**

A simple and a source-container-specific module filter in the `n4js` section of a package.json file.

```
"moduleFilters": {
  "noValidate": [
    "abc*",
    {
      "module": "xyz*",
      "sourceContainer": "src/n4js"
    }
  ],
  "noModuleWrap": [
    // syntax same as for noValidate above
  ]
}
```

<b>mainModule</b>	(string) A plain module specifier defining the project's 'main module'. If this property is defined, other projects can import from this project using imports where the string following keyword <code>from</code> states only the project name and not the complete module specifier (see <a href="#">Import Statement Semantics</a> ). If this property is defined, top-level property <code>main</code> will be ignored.
-------------------	--

<b>testedProjects</b>	(array) List of N4JS components being tested by this project. Only components of project type "test" may declare this property. Furthermore, the referenced projects must all be of the same project type and must not be of type "test" themselves.
<b>implementationId</b>	(string) If this property is defined, this component is called an "implementation project" and the string value provides a unique identifier for the implementation provided in this component. If this is defined, property "implementedProjects" must be defined as well. For details, see <a href="#">Section 12.10, "API and Implementation Components"</a> .
	Only projects of type "application", "processor", "library", "api" or "validation" may declare this property.
<b>implementedProjects</b>	(array) A list of API components (components of type "api") that are implemented by this component. If this is defined, property "implementationId" must be defined as well. For details, see <a href="#">Section 12.10, "API and Implementation Components"</a> . Only components of type "application", "processor", "library", "api" or "validation" may declare this property.
<b>requiredRuntimeLibraries</b>	(array) The list of required runtime library components that are required for the execution of this component. All components but components of type "runtime environment" may declare this property. Each required runtime library must also be specified as a dependency using one of the top-level properties <code>dependencies</code> or <code>devDependencies</code> .
<b>moduleLoader</b>	(string) This property defines what module loaders are supported by the modules in this component. Possible values are the following:
	<b>n4js</b> (default) The modules in this component can be loaded with SystemJS or with CommonJS.
	<b>commonjs</b> Modules in this component must be loaded with CommonJS. When these modules are referenced in generated code (i.e. when importing from these modules), the module specifier will be prefixed with <code>@@cjs/</code> .
	<b>node_builtin</b> Modules in this component represent node built-in modules such as fs or https. When these modules are referenced in generated code (i.e. when importing from these modules), the module specifier will be prefixed with <code>@node/</code> .
<b>extendedRuntimeEnvironment</b>	(string) The name of the runtime environment project that is extended by this component. Only components of type "runtime environment" may declare this property.
<b>providedRuntimeLibraries</b>	(array) The list of runtime library components that are provided by this component. Only components of type "runtime environment" may declare this property.
<b>initModules</b>	(array) A list of modules in terms of module specifiers (strings), that are executed when the component is initialized. This property may only be declared by components of type "runtime environment" and "runtime library".  These init modules are executed right before the execution of a user-specified module starts. They may be used to initialize polyfills or perform other initialization work with regard to the execution environment (e.g. define global properties).
<b>execModule</b>	(string) The module specifier of the exec module of a runtime environment. This property may only be declared by components of type "runtime environment" and "runtime library". The exec module of a runtime environment is executed as the entry-point to the execution of a user-specified module. The underlying script must interpret the execution data and trigger the actual module execution in the runtime environment.
<b>definesPackage</b>	(string) The name of the package this component provides type definitions for. Only components of project type "definition" may declare this property.

All properties described above are optional. The following default values apply:

Property	Default Value
name	name of the folder containing the <code>package.json</code> file
version	"0.0.1"
projectType	"plainjs"
vendorId	"vendor.default"
mainModule	"index"
output	". "

sources

a single source-container of type "source" with path "."

All other properties are undefined if not given in the `package.json` file.

#### Example 12.2. A package.json file with N4JS-specific properties.

The following example illustrates how to use the N4JS-related package.json properties.

```
{
  "name": "SampleProject",
  "version": "0.0.1",
  "author": "Enfore AG",
  "main": "./src/js/main.js",
  "dependencies": {
    "OtherProject": ">=1.2.3 <2.0.0",
    "n4js-runtime-es2015": "latest"
  },
  "devDependencies": {
    "org.eclipse.n4js.mangelhaft": "latest"
  },
  "n4js": {
    "projectType": "library",
    "vendorId": "org.eclipse.n4js",
    "vendorName": "Eclipse N4JS Project",
    "output": "src-gen",
    "mainModule": "a/b/Main",
    "sources": {
      "source": [
        "src/n4js",
        "src/n4js-gen"
      ],
      "external": [
        "src-ext"
      ],
      "test": [
        "src-test"
      ]
    },
    "moduleFilters": {
      "noValidate": [
        "abc*",
        {
          "module": "xyz*",
          "sourceContainer": "src/n4js"
        }
      ],
      "noModuleWrap": [
        // syntax same as for noValidate above
      ],
      "requiredRuntimeLibraries": [
        "n4js-runtime-es2015"
      ]
    }
  }
}
```

### 12.4.3. Constraints

The following constraints apply.

#### GeneralConstraints

##### Requirement: IDE-158: GeneralConstraints (ver. 1)

1. There must be an output directory specified so the compiler(s) can run.

#### Paths

##### Requirement: IDE-159: Paths (ver. 1)

Paths Paths are constrained in the following way:

1. A path cannot appear more than one time within a source fragment type (same applies to paths in the resources section).
2. A path cannot be used in different source fragment types at same times.
3. A path can only be declared exclusively in one of the sections Output, Libraries, Resources or Sources.

4. A path must not contain wild cards.
5. A path has to be relative to the project path.
6. A path has to point to folder.
7. The folder a defined path points to must exist in the project (but in case of non-existent folders or source fragments, only a warning is shown).

### Module Specifiers

**Requirement: IDE-160: Module Specifiers (ver. 1)**

Module Specifiers are constrained in the following way:

1. Within a module filter type no duplicate specifiers are allowed.
2. A module specifier is by default applied relatively to all defined source containers, i.e. if there src and src2 defined as source containers in both folders files are looked up that matches the given module specifier
3. A module specifier can be constrained to be applied only to a certain source container.
4. A module specifier is allowed to contain wildcards but it must resolve to some existing files in the project

### Module Specifier Wildcard Constraints

**Requirement: IDE-161: Module Specifier Wildcard Constraints (ver. 1)**

. All path patterns are case sensitive. . `**` all module specifiers will be matched. . `**/*` all module specifiers will be matched. . `test/A??` matches all module specifiers whose qualified name consists of two segments where the first part matches test and the second part starts with an `A` and then two more characters. . `**/test/**/XYZ` - matches all module specifiers whose qualified name contains a segment that matches test and the last segment ends with an 'XYZ'. . A module specifier wild card isn't allowed to contain `***` . . A module specifier wild card isn't allowed to contain relative navigation. . A module specifier wild card shouldn't contain the file extension (only state the file name (pattern) without extension, valid file extensions will then be used to match the file).

Examples of using external source fragments and filters are given in [Section 13.2.3, "Implementation of External Declarations"](#), see [Example 13.1, "External Definitions and Their Implementations"](#).

### Dependencies to Definition Projects

**Requirement: GH-821001: Dependencies to Definition Projects (ver. 1)**

. For each listed project dependency of type "definition", a corresponding dependency (in the (dev)dependencies section) must be declared, whose "name" matches the "definesPackage" property value of the definition project.

## 12.5. Support for NPM Scopes

NPM supports a namespace concept for npm packages. Such namespaces are called "scopes". For details see <https://docs.npmjs.com/misc/scope> and <https://docs.npmjs.com/getting-started/scoped-packages>. In N4JS, this is supported too.

Terminology:

1. A project's *plain project name* is its name without mentioning the project's scope (if any), e.g. `myProject`.
2. A project's *scope name* is the name of the npm scope a project resides in, including a leading `@`. E.g. `@myScope`.
3. A project's *N4JS project name* is its plain project name, prefixed by its scope name (if any), separated by a `/`. For unscoped projects, this is identical to the plain project name. E.g., `myProject` (if unscoped), `@myScope/myProject` (if scoped).
4. A project's *Eclipse project name* is an ancillary name used only within the Eclipse UI for the project in the workspace. It is equal to the N4JS project name, except that `:` instead of `/` is used as separator between the scope and plain project name. E.g., `myProject` (if unscoped), `@myScope:myProject` (if scoped).

In case the intended meaning is apparent from the context, the "N4JS project name" can simply be referred to as "project name" (as is common practice in the context of npm).

In N4JS, when importing from a module `M` contained in a scoped project `@myScope/myProject`, the import statement's module specifier should have one of the following forms:

- `import * as N from "a/b/c/M";`
- `import * as N from "@myScope/myProject/a/b/c/M";`
- `import * as N from "@myScope/myProject";` (if `M` is specified as main module in `myProject's package.json)

Thus, the N4JS project name, which includes the scope name, is simply used in place of an ordinary, non-scoped project's name. This is in line with conventions in Javascript.

#### General Constraints

##### Requirement: GH-1026: General Constraints (ver. 1)

1. The name given in the package.json file (i.e. value of top-level property "name") must be equal to the project's "N4JS project name", as defined above.
2. The name of the project folder on disk (i.e. folder containing the package.json file) must be equal to the project's "plain project name", as defined above.
3. Iff the project is scoped, this project folder must have a parent folder with a name equal to the project's "scope name", including the leading @ .
4. Within Eclipse, the name of an N4JS project in the workspace UI must be equal to the project's "Eclipse project name", as defined above.

## 12.6. Component Dependencies

There are several dependencies between components. We can distinguish between *require* dependencies and *provide* dependencies.

**require** N4JS Components require:

- APIs
  - RuntimeLibraries and
  - Libraries
- provide**
- Runtime Environments provide Runtime Libraries and maybe extend other Runtime Environments (which means they provide the same runtime libraries as the extended environments and the same base types).
  - SysLibs implement (provide implementations of) APIs

### 12.6.1. Runtime Environment Resolution

In order to execute (run, debug, or test) an *N4JS Component*, an actual *runner* has to be determined. Since runners support runtime environments, this basically means calculating runtime environments which provide all necessary runtime libraries needed by the component. This is done by computing the transitive closure of required runtime libraries and by comparing that with the transitive closure of runtime libraries provided by an environment.

#### Transitive Component Dependencies

##### Definition: Transitive Component Dependencies

We defined the following transitive closures of dependencies:

For a given N4JS Component  $C$  we define

$$\begin{aligned} C.requiredAPIS^* &= C.requiredPIs \\ &\cup \\ &\bigcup_{L \in C.requiredLibs} C.requiredPIs^* \\ &\cup \\ &\bigcup_{S \in C.requiredAPIS} S.requiredAPIS^* \\ C.requiredLibs^* &= C.requiredRTLibs \\ &\cup \\ &\bigcup_{L \in C.requiredLibs} L.requiredRTLibs^* \end{aligned}$$

For a given Runtime Environment  $E$  we define

$$\begin{aligned} E.providedRTLibs^* &= E.providedRTLibs \\ &\cup \\ &\bigcup_{ES \in E.extendedRTEnv} ES.E.providedRTLibs^* \end{aligned}$$

#### Runtime Environment Resolution

##### Requirement: IDE-162: Runtime Environment Resolution (ver. 1)

An N4JS Component  $C$  can be executed in an runtime environment  $E$ , written as  $E \models C$  if the following constraints hold:

1. The environment must provide all runtime libraries transitively required by the component:

$$\forall R \in C.\text{requiredRTLibs}^*: R \in E.\text{providedRTLibs}^*$$

2. There exist libraries which can be executed by the environment, so that all APIs are implemented:

$$\forall S \in C.\text{requiredAPIs}^* \exists I: S \in I.\text{ImplementedAPIs} \wedge E \models I$$

## 12.7. N4JS Type Definitions

N4JS projects can depend on ordinary JavaScript projects by including them in the package.json file. From there on, modules of those JavaScript projects can be imported when writing N4JS code. However, since JavaScript is untyped there will not be any type information for e.g. classes, functions of ordinary JavaScript projects unless this type information is provided by a type definition project. Type definition projects do only contain n4jsd files that reflect the classes and functions of a specific npm. To refer to a JavaScript npm, the term *plain-JS project* will be used.

### 12.7.1. Specify Type Definition

A type definition project is structured like a normal npm. The major difference is that it provides n4jsd files instead of js files. These n4jsd files are named like and located at the exact position in the file tree as their js-counterparts. This ensures the type definition module and the corresponding plain-JS module to have the same fully-qualified name. Besides the usual properties the package.json file usually needs to specify the following properties in the n4js section.

#### Package.json: Important properties for type definition projects

```
{
  "n4js": {
    "projectType": "definition"
    "definesPackage": "..."
    "mainModule": "..."
  }
}
```

The project type declares this project to be a type definition projects. Consequently, it has to also declare the name for which plain-JS project its type definitions are provided (using `definesPackage`). Lastly, the main module has to be specified since this information will not be taken from the package.json of the plain-JS project. In case the plain-JS project does not have a main module or the main module is located at the package root, the `mainModule` property can be omitted.

### 12.7.2. Name Conventions

A type definition package can have an arbitrary name and define an arbitrary npm package. This can be handy for testing purposes or just creating some temporary type definitions for a local package. However, we chose to use a convention to simplify finding the right type definition package for a specific plain-JS project. First, the scope `@n4jsd` and second the exact name of the plain-JS project is used. For instance, when a user wants to install type definitions for the plain-JS project `express`, our related type definitions are called `@n4jsd/express`.

### 12.7.3. Version Conventions

Since the plain-JS project will evolve over time and publish different versions, the need arises to also publish the type definition project in different versions that reflect this evolution. In addition to the evolution of the plain-JS project, a new version of the type definition project can also become necessary in case a bug in the type definitions was found or in case the language of N4JS changes and the type definitions have to be adjusted accordingly. Effectively, this will lead to a situation where both the implementation and the type definition project have a version that are technically unrelated from an npm point of view, but still are somehow related to each other from a semantical point of view. To keep the distinct versioning of both of the projects manageable, we propose the following conventions to partially align the type definition project's version to that of the plain-JS project.

#### 12.7.3.1. Define a New Type Definition Package

We use the following convention to compute the version of type definition packages.

$$\text{Major}_{\text{types}}.\text{Minor}_{\text{types}}.\text{Patch}_{\text{types}} = \text{Major}_{\text{impl}}.\text{Minor}_{\text{impl}}.0$$

$$\text{Major}_{\text{types}}.\text{Minor}_{\text{types}}.\text{Patch}_{\text{types}} = 3.3.0$$

Let's say that a new version of a type definition package called `types` should be created that defines types for an npm called `impl` of version `Majorimpl.Minorimpl.Patchimpl`. According to our convention, the major and minor version numbers of the type definition package should just be copied from the version of the `impl` package. However, the version patch number of `types` should not be taken from `impl`. Instead, the patch number of `types` starts with `0` and increases with every update of this type definition version. For instance when a bug was found in version `Majortypes.Minortypes.0`, the definitions have been extended, or adjusted to new language features, only the patch number increases to e.g. `Majortypes.Minortypes.1`.

### 12.7.3.2. Using a Type Definition Package

On the client side, a type definition package is listed among the dependency section. Here we use the following convention to specify the required version of a type definition package.

```
"dependencies": {
    "@n4jsd/Types": "<=Majorimpl.Minorimpl.*"
}

"dependencies": {
    "express": "^3.3.3",
    "@n4jsd/express": "<=3.3.*"
}
```

According to this convention, the major and minor version numbers of the implementation package are used, prepended with a smaller-equals and appended with an asterisk for the patch number. This also applies when the implementation version contains a tilde, a caret, etc., or is omitting a minor or patch number. In case a non SemVer version is given (e.g. `latest`, empty string, url, etc.) it is recommended to plain copy the non SemVer version when possible.

### 12.7.3.3. Rational

The rational behind this convention reflects the idea of semantic versioning:

Given a version number MAJOR.MINOR.PATCH, increment the:

- MAJOR version when you make incompatible API changes,
- MINOR version when you add functionality in a backwards-compatible manner, and
- PATCH version when you make backwards-compatible bug fixes.

— [semver.org](http://semver.org)

Patch version increments are always backwards compatible according to SemVer. In addition also no further functionality is added since this would imply at least an increment of the minor version. Consequently, patch versions do not affect the interface or type information of an plain-JS project. This is why patch version number fully suffices to reflect bug fixes and language changes for a given major.minor version.

On client side, we recommend to use a smaller-equals qualifier because most probably there will not be the exact version of a requested type definition project. Instead, only some major.minor versions will have a type definition counterpart. Using a smaller-equals qualifier will make sure that a client will always get the latest version of a requested plain-JS project version. In case a newer version of the plain-JS project was already published, this convention guarantees that a compatible version of the type definition project is installed.

## 12.8. Modules

All N4JS files are modules, sometimes also called compilation unit (CU). This is the overall structure of a module, based on [\[ECMA15a\(p.S14\)\]](#).

```
Script: {Script}
    annotations+=ScriptAnnotation*
    scriptElements+=ScriptElement*;

/*
 * The top level elements in a script are type declarations, exports, imports or statements
 */
ScriptElement:
    AnnotatedScriptElement
    | N4ClassDeclaration<Yield=false>
    | N4InterfaceDeclaration<Yield=false>
    | N4EnumDeclaration<Yield=false>
    | ImportDeclaration
    | ExportDeclaration
    | RootStatement<Yield=false>
;
```

Grammar and semantics of import statement is described in [Section 9.2.4, “Import Statement”](#); of export statement described in [Section 9.2.5, “Export Statement”](#).

An import statement imports a variable declaration, function declaration, or N4 type declaration defined and exported by another module into the current module under the given alias (which is similar to the original name if no alias is defined). The name of the module is its project’s source folder’s relative path without any extension, see [Section 3.2.3, “Qualified Names”](#) for details.

This are the properties of script, which can be specified by the user:

**annotations** Arbitrary annotations, see [Chapter 10, Annotations](#) and below for details.

**scriptElements** The content of the script.

And we additionally define the following pseudo properties:

**path** File system path (path delimiter is always `'/'`) relative to the source fragment of the file without the extension.  
E.g.: given a source folder `src`, `path` of a module located at:

- `src/n4/lang/List.js` is `n4/lang/List`
- `src/n4/lang/Objects.prototypes` is `n4/lang/Objects`

**expandedPath** Pseudo property consists of the project name and project version of the module followed by the path, the concrete syntax is: `<project.name>-<project.version>/<module.path>` where project version includes all version parts except the qualifier.

E.g. given a module with path `n4/lang/List` in a project `lib` with version `1.0.0`, the `expandedPath` is `lib-1.0.0/n4/lang/List`.

**loadtimeDeps** Pseudo property contains all load time dependencies of this module.

**runtimeDeps** Pseudo property contains all runtime dependencies of this module.

**allDeps** Pseudo property contains all dependencies of this module. This is the union of `loadtimeDeps` and `runtimeDeps` which maintains the ordering of both lists, with the `loadtimeDeps` at the front.

Pseudo properties to be set via annotations are explained in [Chapter 10, Annotations](#).

## 12.9. Properties Files

Properties files have the file extension `properties` and describe how to localize text in a project. They basically define keys [Section 4.6.4, “Primitive Pathselector and I18nKey”](#) with their values. The key is used during runtime to retrieve text localized to the user's locale.

### 12.9.1. Syntax

The syntax of a resource file is defined as:

```
ResourceFile: Comment* | $entry+$ Entry*;
Comment: 'pass:[#]' .* EOL;
Entry: $key$ = KeyIdentifier '=' $value$ = .* EOL;
KeyIdentifier: LETTER (DIGIT | LETTER | '.')*;
```

### 12.9.2. Constraints

Properties files have to be stored in source fragment of type source. The *base folder* for storing the properties files of a project *p* is `p.name/nls`. The language-specific resource files are stored in subfolders of the base folder. The base language (normally english) has to be located in a subfolder of the base folder. The resource files for other languages have to be located in a subfolder with the name given by syntax `<ISO Language Code>-<ISO Country Code>`, where ISO Language Code is given by the ISO-639 standard and ISO Country Code is given by the ISO-3166 standard.

All resource files stored in a language folder are compiled to a JavaScript file which exports all resource keys as an object literal.

The resource files of a project are automatically loaded. To access a resource key `key` stored in a resource file `my.properties`, you have to use the file name as a prefix (e.g. you have to use the key `my.key`).

## 12.10. API and Implementation Components

Instead of providing an implementation, N4JS components may only define an API by way of one or more n4jsd files which is then implemented by separate implementation projects. For one such API project, several implementation projects may be provided. Client code using the API will always be bound to the API project only, i.e. only the API project will appear in the client project's `package.json` file under `dependencies`. When launching the client code, the launcher will choose an appropriate implementation for each API project in the client code's direct or indirect dependencies and transparently replace the API project by the implementation project. In other words, instead of the API project's output folder, the implementation project's output folder will be put on the class path. Static compile time validations ensure that the implementation projects comply to their corresponding API project.

Note how this concept can be seen as an alternative way of providing the implementation for an n4jsd file: usually n4jsd files are used to define types that are implemented in plain JavaScript code or provided by the runtime; this concept allows for providing the implementation of an n4jsd file in form of ordinary N4JS code.

At this time, the concept of API and implementation components is in a prototype phase and the tool support is limited. The goal is to gain experience from using the early prototype support and then refine the concept over time.

Here is a summary of the most important details of this concept (they are all subject to discussion and change):

- Support for this concept, esp. validations, should not be built into the core language but rather implemented as a separate validation/analysis tool.
- Validation is currently provided in the form of a separate view: the API / Implementation compare view.
- A project that defines one or more other projects in its `package.json` file under `implementedProjects` (cf. `implementedProjects`) is called *implementation project*. A project that has another project pointing to itself via `ImplementedProjects` is called *API project*. Note that, at the moment, there is no explicit definition making a project an API project.
- An implementation project must define an implementation ID in its `package.json` file using the `implementationId` property in the `n4js` section (cf. `implementationId`).
- For each public or `public@Internal` classifier or enum in an API project, there must be a corresponding type with the same fully-qualified name of the same or higher visibility in the implementation project. For each member of such a type in the API, there must exist a corresponding, owned or inherited type-compatible member in the implementation type.
- Beyond type compatibility, formal parameters should have the same name on API and implementation side; however, different names are legal but should be highlighted by API / Implementation tool support as a (legal) change.
- Comments regarding the state of the API or implementation may be added to the JSDoc in the source code using the special tag `@apiNote`. API / Implementation tool support should extract and present this information to the user in an appropriate form.
- If an API class `C` implements an interface `I`, it has to explicitly (re-) declare all members of `I` similar to the implementation. This is necessary for abstract classes anyway in order to distinguish the implemented methods from the non-implemented ones. For concrete classes, we want all members in `C` in order to be complete and avoid problems when the interface is changed or `C` is made abstract.

### 12.10.1. Execution of API and Implementation Components

When launching an N4JS component `C` under runtime environment `RE`, the user may(!) provide an implementation ID `IID` to run. Then, for each API project `A` in the direct or indirect dependencies of `C` an implementation project is chosen as follows:

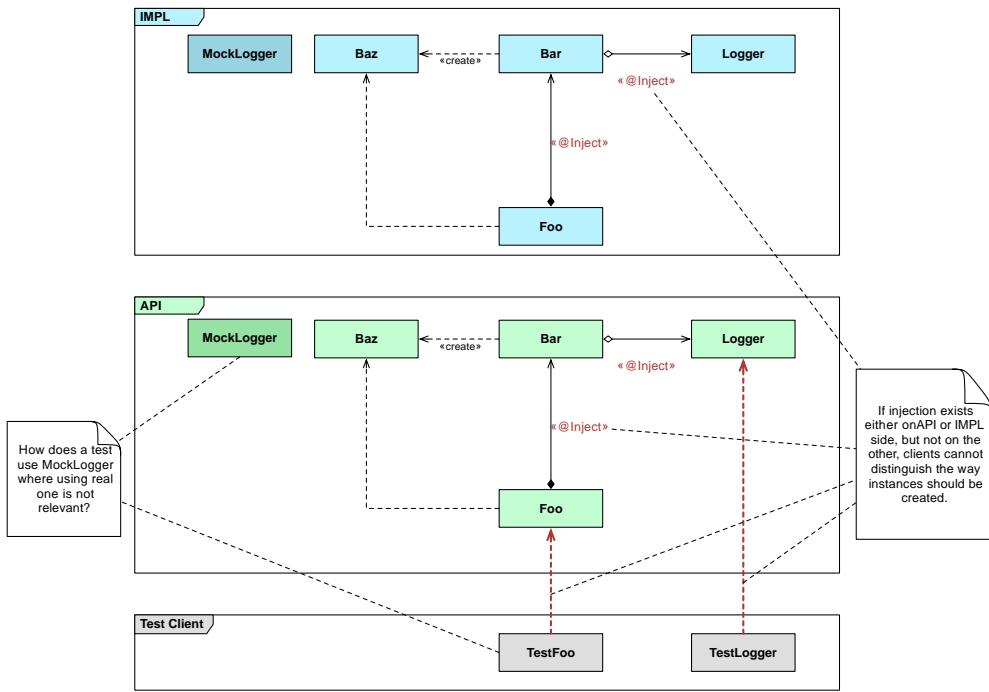
1. Collect all implementation projects for `A` (i.e. projects that specify `A` in their `package.json` file under `implementedProjects`).
2. Remove implementation projects that cannot be run under runtime environment `RE`, using the same logic as for running ordinary N4JS components (this step is not implemented yet!).
3. If there are no implementation projects left, show an error.
4. If `IID` is defined (i.e. user specified an implementation ID to run), then:
  - a. If there is an implementation project left with implementation ID `IID`, use that.
  - b. Otherwise, show an error.
5. If `IID` is undefined, then
  - a. If there is exactly 1 implementation project left, use it.
  - b. Otherwise, in UI mode prompt the user for a choice, in headless mode show an error.

Having found an implementation project `In` for each API project `An`, launch as usual except that whenever `An`'s output folder would be used, use `In`'s output folder (esp. when constructing a `class path`) and when loading or importing a type from `An` return the corresponding type with the same fully-qualified name from `In`.

### 12.11. API and Implementation With DI

API projects may use N4JS DI (Section 11.2, “Dependency Injection”) language features which require Implementation projects to provide DI-compatible behaviour in order to allow a Client (implemented against an API project) to be executed with a given Implementation project. This is essential for normal execution and for test execution.

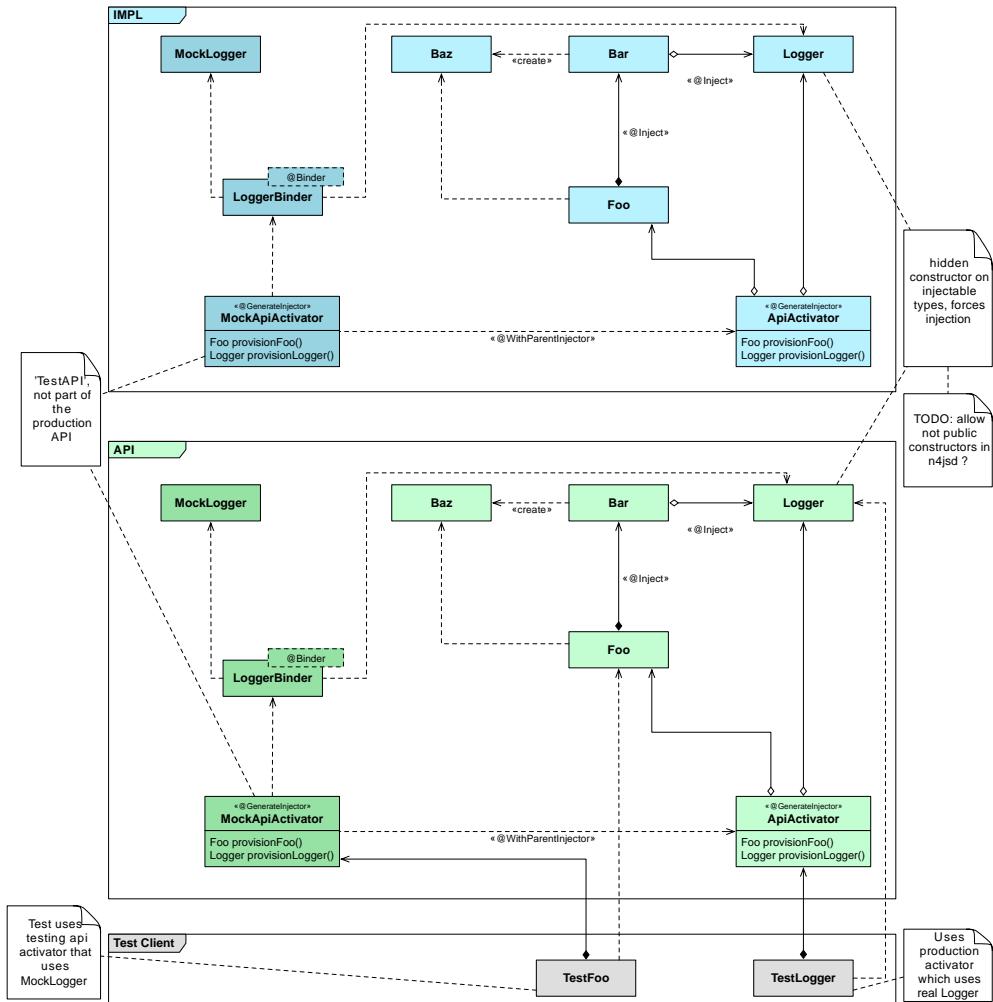
Figure 12.3, “Overview of API tests with DI” shows some of those considerations from test client point of view.



**Figure 12.3. Overview of API tests with DI**

Static DI mechanisms in N4JS allow an API project to enforce Implementation projects to provide all necessary information. This allows clients to work seamlessly with various implementations without specific knowledge about them or without relying on extra tools for proper project wiring.

Figure 12.4, “API tests with static DI” shows how API project defines project wiring and enforces certain level of testability.



**Figure 12.4.** API tests with static DI

During Client execution, whether it is test execution or not, N4JS mechanisms will replace the API project with a proper implementation project. During runtime DI mechanisms will take care of providing proper instances of implantation types.

Figure 12.5, “Types view and Instances view” shows Types View perspective of the client, and Instances View perspective of the client.

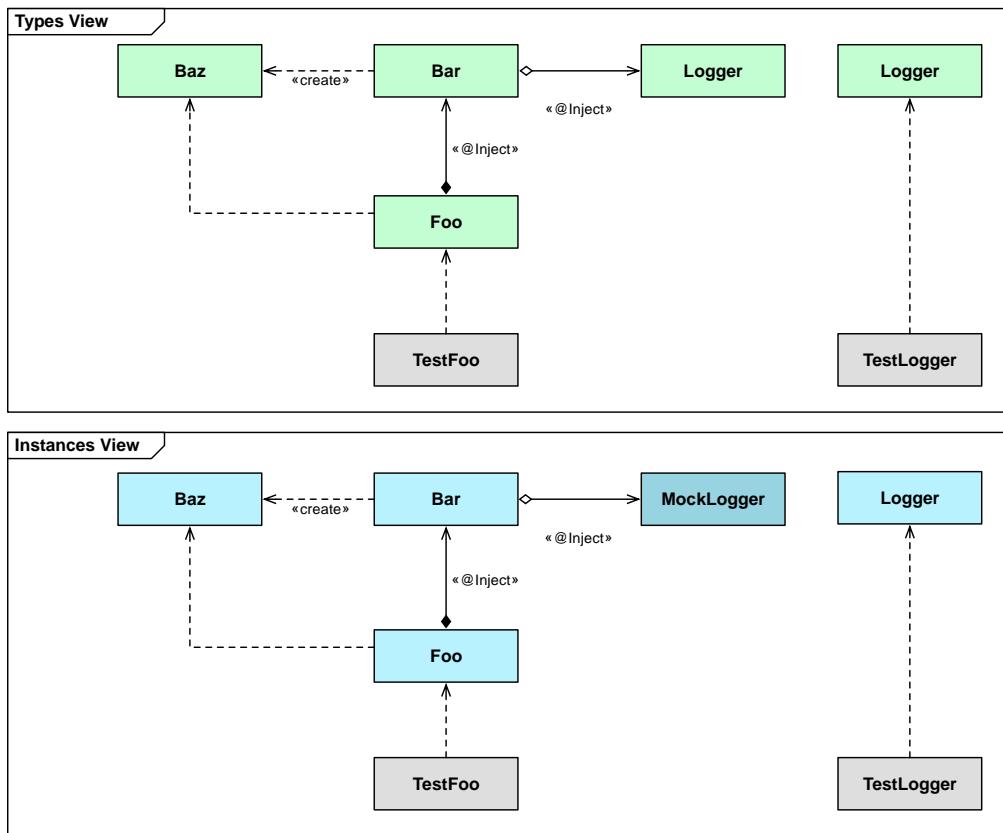


Figure 12.5. Types view and Instances view

# Chapter 13. PlainJS

Since N4JS is a super set of JavaScript, is it both possible to use plain JavaScript in N4JS and vice versa. There may be some obstacles due to concepts introduced by N4JS to make code more maintainable and robust:

1. N4JS' static type system may complain about some older JavaScript hacks. Declared types, in particular, are assumed to be implicitly frozen.
2. In N4JS, modules are used as namespaces with explicit export and import statements. The notion of globals is not directly supported by N4JS as this leads to unexpected side effects (several components providing and thus overriding global definitions, for example).
3. N4JS defines a (ECMAScript 6 compatible) concept of object-oriented programming which may conflict with other plain JavaScript solutions.

To overcome these problems, N4JS provides a couple of techniques summarized in this chapter.

## 13.1. Type Inference and Validation for Plain JS

In plain JavaScript mode:

1. All declared variables are inferred to `any+`.
2. All declared functions return and accept a variadic number of arguments of type `any+`.
3. It is allowed to use the `return` statement with or without an expression.
4. New expressions with a receiver of `any+` is allowed.
5. No type arguments are required for generic built-in types.
6. Assigning a value to a read-only variable is not checked.
7. Undeclared variables are treated as `any+` as well.

Note that this essentially disables all validation particularly since methods such as the 'import'-like function `require` are unknown.

## 13.2. External Declarations

N4JS supports declaring external classes as a means to declare classes whose implementation is not N4JS so they can be used from N4JS. Together with structural typing, this allows N4JS to seamlessly integrate frameworks and libraries which have not been implemented in N4JS but in plain ECMAScript or another language.

### External allowed occurrences

#### Requirement: IDE-163: External allowed occurrences (ver. 1)

- Declarations with external flags are only allowed in files with the extension `n4jsd` (so called N4JS definition files).
- Only external classes, external interfaces marked with `@N4JS`, external enums, external function declarations and structurally typed interfaces are allowed in a `n4jsd` file.
- Declarations with external flags are allowed to subclass built-in type `Error` type and all of its descendants such as `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` and `URIError`, although any of the error types are annotated with `@N4JS`.

The following explanations apply to all external declarations except where stated otherwise.

In general, an external declaration uses the same syntax as the declaration of a normal N4JS declaration with the addition of the modifier `external`.

External classifiers are always 'entirely external' in that it is not possible to combine defined methods and external methods within a single class or interface.

### External classes inheritance

#### Requirement: IDE-164: External classes inheritance (ver. 1)

1. An external class *without* the `@N4JS` annotation can only inherit from another external class or from one of the built-in ECMAScript types (e.g. `Object`). That is, by default external classes are derived from `Object`. 2. An external class *with* the annotation `@N4JS` can only inherit from another external class annotated with `@N4JS` or from non-external N4JS classes.

### Structurally typed interface implementation

**Requirement: IDE-165:** Structurally typed interface implementation (ver. 1)

1. An external class *without* the annotation `@N4JS` can only be implemented by structurally typed interfaces. 2. An external class *with* the annotation `@N4JS` can only be implemented by structurally typed interfaces annotated with `@N4JS`. 3. An external interface *without* the annotation `@N4JS` must be defined structurally.

### External interface inheritance

**Requirement: IDE-166:** External interface inheritance (ver. 1)

1. An interface in a `n4jsd` file *without* the annotation `@N4JS` can only inherit from another interface within a `n4jsd` file.
2. An interface *with* the `@N4JS` annotation can only inherit from another interface annotated with `@N4JS`.

### External class/interface members

**Requirement: IDE-167:** External class/interface members (ver. 1)

1. The static and instance methods, getters and setters of an external class must not have a method body.
2. The static and instance fields of an external class must not have an initializer.
3. The constructor of an external class without the annotation `@N4JS` must not be declared private.
4. Methods in interfaces with default implementation which cannot be expressed in the definition file must be annotated with `@ProvidesDefaultImplementation`. This is only allowed in interfaces annotated with `@N4JS`.
5. Fields in interfaces or classes with initializers which cannot be expressed in the definition file, must be annotated with `@ProvidesInitializer`. This is only allowed in classes or interfaces annotated with `@N4JS`.

This means that in external classes, all members except constructors may be declared private even if the class is not annotated with `@N4JS`. In interfaces, however, private members are disallowed anyway, cf. [Req-IDE-48].

### Other external declarations

**Requirement: IDE-168:** Other external declarations (ver. 1)

1. The literals of an external enum must not have a value.
2. An external function declaration must not have a body.

## 13.2.1. Declaring externals

By default, the implicit supertype of an external class is `Object`. If the `@N4JS` annotation is provided it is `N4Object`. If a superclass is explicitly given, the constraints from the previous section apply.

## 13.2.2. Instantiating external classes

In most cases, it is desirable to instantiate external classes from external projects. Publicly exporting the class definition and providing a public constructor is good practice.

In some cases, the instantiation from an outer scope is not wanted. A possible approach is to use a structurally typed interface instead of a class to link to the implementation.

In case of API-definitions (see Section 12.10, “API and Implementation Components”), it might be useful to limit the visibility of classes to narrower scopes such as package or private.

External declarations can be instantiated if the following three requirements are fulfilled (not a constraint!):

- External declarations have to be exported and be marked as public so they are accessible from outside.
- The contained or inherited constructor of an external class must be public.
- The external class must be linked to an implementation module (see below Section 13.2.3, “Implementation of External Declarations”).

### 13.2.3. Implementation of External Declarations

All external declarations must be associated with an external implementation module in one way or another. Any time the external declaration is imported, the compiler generates code that imports the corresponding implementation module at runtime.

There are two possible ways of linking an external declaration to its corresponding implementation:

1. By naming convention defined in the `package.json` file.
2. By declaring that the implementation is provided by the JavaScript runtime, see [Section 13.4, “Runtime Definitions”](#) for details.

The naming convention is based on the `external` source fragments defined in the `package.json` file ([Package.json File](#)). If the implementation is provided by the runtime directly, then this can be also specified in the `package.json` file by a module filter.

The implicit link via the naming convention is used to link an external class declaration to its non-N4JS implementation module. It does not effect validation, but only compilation and runtime. Essentially, this makes the compiler generate code so that at runtime, the linked implementation module is imported instead of the declaration module.

In most use cases of external declarations you also want to disable validation and module wrapping by specifying appropriate filters in the `package.json` file.

Occasionally it is not possible for the validation to correctly detect a corresponding implementation element. For that reason, it is possible to disable validation of implementations completely via `@@IgnoreImplementation`.

#### Implementation of External Declarations

##### Requirement: IDE-169: Implementation of External Declarations (ver. 1)

For a given external declaration  $D$  but not for API-definitions<sup>1</sup>, the following constraints must hold:

1. If the declaration is neither provided by runtime nor validation of implementation is disabled, a corresponding implementation must be found by the naming convention. If no such implementation is found, a warning is generated.

#### Example 13.1. External Definitions and Their Implementations

If, in addition to standard `source`, the `source-external` fragment is provided in `Sources`, `n4jsd` files in the folder tree below source folders will be related to modules of the same name in the external folders. This is shown in [Figure 13.1, “External Class Implementation, Naming Convention”](#).

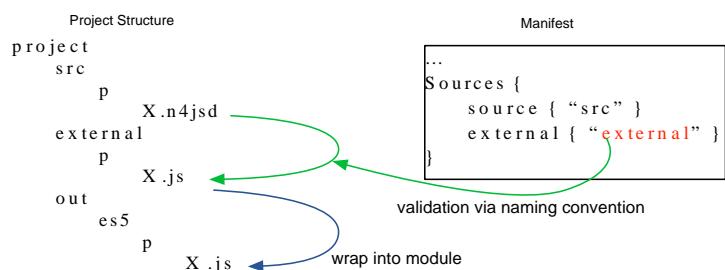


Figure 13.1. External Class Implementation, Naming Convention

### 13.2.4. Example

Assume the following non-N4JS module:

#### Example 13.2. External Classes

```

module.exports = {
  "Point": function Point(x, y) {
    this.x = x;
    this.y = y;
  },
  "Circle": function Circle(center, radius) {
    this.center = center;
    this.radius = radius;
  }
};
  
```

<sup>1</sup> [Section 12.10, “API and Implementation Components”](#)

```

    this.scaleX = function(x){ this.x = x; }
this.scaleY= function(y){ this.y = y; }
}
}

```

Assuming

- `shapes.js` is placed in project folder `/external/a/b/shapes.js`
- `shapes.n4jsd` is placed in project folder `/src/a/b/shapes.n4jsd`
- `package.json` defines `src` as source folder and `external` as external source folder

the following N4JS external class declarations in `shapes.n4jsd` are sufficient:

```

export external public class Point {
    x: number; y: number;
    constructor(x: number, y: number);
}

export external public class Circle {
    center: Point; radius: number;
    constructor(center: Point, radius: number);
}

```

Note that the class and interface names in n4jsd files must match those in the js files, respectively.

#### Example 13.3. Structurally-typed external interfaces

```

export external public interface ~Scalable {
    scaleX(factor: number);
    scaleY(factor: number);
}

export external public class Circle implements Scalable {
    center: Point;
    radius: number; x: number; y: number;

    @Override public scaleX(factor: number);
    @Override public scaleY(factor: number);

    constructor(center: Point, radius: number);
}

```

### 13.3. Global Definitions

Existing JavaScript libraries and built-in objects provided by certain JavaScript environments often globally define variables. Although it is not recommended to use global definitions, this cannot always be avoided.

N4JS supports global definitions via the annotation `Global`. This annotation can only be defined on modules (via `@@Global`) – this means that all declarations in the module are globally defined.<sup>2</sup>

We introduce a new pseudo property `global` on all declared elements accordingly:

`@Global` Boolean flag set to true if annotation `@Global` is set in containing module. The flag indicates that the exported element is globally available and must not be imported.

Since definition of global elements is not supported by N4JS directly, this can be only used in external definitions. A declaration with `global` can be used without explicit import statement. It is not possible to import these declarations.

#### Global Definitions

##### Requirement: IDE-170: Global Definitions (ver. 1)

#### Global Definitions

For a declaration `D` with `D.global = true`, not a polyfill (`D.polyfill = false`), the following constraints must hold:

1. The name of the definition must not be equal to any primitive type (`string`, `number` etc.), `any`, or an built-in N4 type (`N4Object` etc.).

<sup>2</sup> Global basically means that the module defines no namespace on its own. Thus the annotation is a script/module related annotation.

2. If the name of the definition equals a basic runtime time Object Type then the project must be a runtime environment:

```
D.name ∈ { 'Object','Function','Array','String','Boolean'      'Number','Math','Date','RegExp','Error','JSON' }
⇒ D.containingProject.type = runtimeEnvironment
```

## 13.4. Runtime Definitions

Some elements are predefined by the JavaScript runtime such as DOM elements by the browser or built-in ECMAScript or non-standard objects. These elements can be defined by means of external definitions; however, no actual implementation can be provided as these elements are actually provided by the runtime itself.

Since these cases are rather rare and in order to enable additional checks such as verification that a given runtime actually provides the elements, this kind of element can only be defined in components of type runtime environment or runtime library (cf Section 12.2.2, “Runtime Environment and Runtime Libraries”).

N4JS supports runtime definitions via the annotation `@ProvidedByRuntime`. This annotation can be defined

- on modules (via `@@ProvidedByRuntime`) – this means that all declarations in the module are provided by the runtime
- on export statements or declarations.

We introduce a new pseudo property `providedByRuntime` accordingly:

`@ProvidedByRuntime` Boolean flag set to true if the annotation `@ProvidedByRuntime` is set. Flag indicates that the element is only declared in the module but its implementation is provided by the runtime.

Since built-in types are usually defined globally, the annotation `@ProvidedByRuntime` is usually used in combination with `@Global`.

### Provided By Runtime

#### Requirement: IDE-171: Provided By Runtime (ver. 1)

For a declaration  $D$  with  $D.providedByRuntime = \text{true}$ , the following constraints must hold:

1. The declaration must either be an export declaration itself or an exportable declaration.
2. The declaration must be contained in a definition module.
3. The declaration must be (indirectly) contained in a component of type `runtimeEnvironment` or `runtimeLibrary`.
4. There must be no implementation file with the same name as the definition module if annotation is defined for a whole module.

## 13.5. Applying Polyfills

(Runtime) Libraries often do not provide completely new types but modify existing types. The ECMA-402 Internationalization Standard [ECMA12a], for example, changes methods of the built-in class `Date` to be timezone-aware. Other scenarios include new functionality provided by browsers which are not part of an official standard yet. Even ECMAScript 6 [ECMA15a] extends the predecessor [ECMA11a] in terms of new methods or new method parameters added to existing types. It also adds completely new classes and features, of course.

The syntax of runtime polyfills is described in section Section 11.4, “Polyfill Definitions”. Here, an example of applying a runtime polyfill is detailed.

#### Example 13.4. `Object.observe` with Polyfill

The following snippet demonstrates how to define a polyfill of the built-in class `Object` to add the new ECMAScript 7 observer functionality. This snippet has to be defined in a runtime library or environment.

```
@ProvidedByRuntime
@Global

@Polyfill
export external public class Object extends Object {
    public static Object observe(Object object, Function callback, Array<string>? accept);
}
```

A client referring to this runtime library (or environment) can now access the observer methods as if it were defined directly in the original declaration of `Object`.

# Chapter 14. JSDoc

In N4JS, comments starting with two asterisks (in `/** ... */`) are interpreted as documentation comments. The format is similar to JavaDoc or Google Closure annotations.

## 14.1. General N4JSDoc Features

We distinguish between line and inline tags. The format of the content of a tag is specific to the tag. Most line tags, however, contain a description which is simply multiline text with nested inline tags. Every comment may start with a description.

### 14.1.1. Provided Inline Tags

#### 14.1.1.1. @code

Small code snippet, not validated yet.

#### 14.1.1.2. @link

Link to a type of element, not validated or supported in navigation yet.

## 14.2. N4JSDoc for User Projects

### 14.2.1. Standard Tags

#### 14.2.1.1. @author

Name of author, repeat for multiple authors. Name of author is not validated.

#### 14.2.1.2. @param

Parameter description, not validated at the moment.

#### 14.2.1.3. @return

Return description, not validated at the moment.

### 14.2.2. Test Related Tags

The following tags are supposed to be used only in tests.

#### 14.2.2.1. @testee

Link to type (maybe a function) or member tested by the test.

##### @testee Semantics

###### Requirement: IDE-172: @testee Semantics (ver. 1)

. Tag can be only used on either .. methods annotated with `@Test` .. classes in test projects or folders, cf. [Section 14.2.2.2, “@testeeFromType”](#). . Tag requires a single argument, which is a fully qualified name to a type, including the module specifier. The format is as follows: `moduleSpecifier '..' typeName ( ('.'|'#') memberName)?` . Tag is *not* repeatable, that is a single test method (or class) can refer to only one testee. . Tag precedes the `reqid` tag, i.e., if a `@testee` is specified, the `reqid` will be ignored.

###### Example 14.1. @testee

The target element is to be fully qualified including the module specifier. The module specifier is simply the source folder relative path name with forward slashes. Type and element are added to that using dot as a separator. For example:

```
/**  
 * @testee some/package/Module.Type.member  
 */
```

### 14.2.2.2. @testeeFromType

Instead of directly linking a test method to a testee, the testee is to be derived from the linked testee of the test class. This is useful if a base test class is defined with generic tests, e.g., for testing methods defined in an interface and implemented by some classes. This base test class is then extended by concrete test classes, correctly setting up the test fixture, e.g., creating instances of some classes implementing the interfaces tested in the base class.

#### Example 14.2. Usage of testeeFromType

In the following example, the is used. This tag will lead to a test documentation for `B.foo` and `C.foo`.

```
abstract class Base {
    /**
     * @testeeFromType
     */
    @Test testFoo() { ... }
}

/**
 * @testee B.foo
 */
class TestB extends Base {}

/**
 * @testee C.foo
 */
class TestC extends Base {}
```



The resulting spec has to be double-checked for consistency since it is easily possible that too many constraints are generated.

### 14.2.2.3. @testeeType and @testeeMember

Specifying the testee at the test method directly should be sufficient for most cases. The `@testeeFromType` tag already provides support for some cases in which a base test class is reused by subtypes. This case usually only works if the base test class tests a single method only. If the base test class tests several methods and if a sub test class only provides a different fixture, this mechanism is not sufficient. For that purpose, the two tags `@testeeFromType` and `@@testeeMember` are to be used. They enable the separation of a test related to a specific member and the concrete receiver type of the tested member.

The `@testeeType` is to be defined in the test class JSDoc (actually, it is not recognized when defined in a member JSDoc). The `@@testeeMember` is specified in the test method JSDoc. The "real" testee is then computed from the testee type and the testee method.



1. This only works for instance members, so far!
2. There is no validation for invalid combinations!

#### Example 14.3. testeeType and testeeMethod

Assume the following testees:

```
class A {
    foo(): void { ... }
    bar(): void { ... this.foo(); ... }
}
class B extends A {
    @Override foo() { ... }
```

Assume that the tests have to ensure the same semantics for `bar`, which is maybe changed by a wrong implementation of `foo`. That is, `bar` is to be tested in case of the receiver type `A` and `B`. This can be achieved by the following tests:

```
/** 
 * @testeeType A.A
```

```

/*
class ATest {
    fixture(): A { return new A(); }

    /**
     * @testeeMember bar
     */
    @Test testBar(): void { assertBehavior( fixture().bar() ); }
}

/**
 * @testeeType B.B
 */
class BTest extends ATest {
    @Override fixture(): B { return new B(); }
}

```

This actually defines two tests, which is also recognized by the spec exporter:

1. `testBar` for a receiver of type `A`:

`ATest`'s JSDoc `@testeeType + ATest.testBar`'s JSDoc `@testeeMember` = testee `A.A.bar`

2. `testBar` for a receiver of type `B`:

`BTest`'s JSDoc `@testeeType + ATest.testBar`'s JSDoc `@testeeMember` = testee `B.B.bar`



In all cases when `@testeeFromType` or `@testeeType / @testeeMember` is used, the resulting spec has to be double-checked for consistency. Consider if the multiplication of spec constraints is truly required, in particular if the original semantics of a method is not changed. Remember: It is possible to write API tests and omit the spec constraint generation simply by not adding the testee links.

#### Example 14.4. `testeeType` and `testeeMethod` with omitted constraints

Assume testees similar as in [Example 14.3, “testeeType and testeeMethod”](#). Since the semantics of `bar` is not changed in `B`, it is probably not necessary to generate the same constraint in the documentation for `bar` twice (one in the section for class `A` and another one in the section of class `B`). Still, we want the test to be executed for both receivers. This is how it is achieved:

```

abstract class BaseTest {
    abstract fixture(): A;

    /**
     * @testeeMember bar
     */
    @Test testBar(): void { assertBehavior( fixture().bar() ); }
}

/**
 * @testeeType A.A
 */
class ATest extends BaseTest {
    fixture(): A { return new A(); }
}

class BTest extends BaseTest {
    @Override fixture(): B { return new B(); }
}

```

This actually defines two tests as in the previous example. Only one constraint is created in the spec by the spec exporter:

1. `testBar` for a receiver of type `A`:

`ATest`'s JSDoc `@testeeType + BaseTest.testBar`'s JSDoc `@testeeMember` = testee `A.A.bar`

Although a test for receiver of type `B` is run, no additional constraint is created since there is no `@testeeType` available neither in `BTest` nor in `BaseTest`.

#### 14.2.2.4. `@reqid` in Tests

ID of feature used in `JSDoc` for the requirements section. If no testee (via one of the tags above) is given, then the test is linked to the requirement with given id.

## 14.3. N4JSDoc for API and Implementation Projects

The following tags are supposed to be used in API and implementation projects.

### 14.3.1. @apiNote

Simple note that is shown in the API compare view.

### 14.3.2. API Project Tags

The following tags are supposed to be used in API projects only.

#### 14.3.2.1. @apiState

State of type or member definition, e.g., stable or draft. This can be used to define a history. In this case, the tag has to be repeated.  
For example:

```
/**  
 * @apiState stable (WK)  
 * @apiState reviewed (JvP)  
 */
```

# Chapter 15. JSX

## 15.1. JSX Support

N4JS scripts with file extension "n4jsx" support the special [JSX syntax](#). Files ending with "jsx" are considered as plain Javascript.

### React Component

#### Requirement: IDE-241101: React Component (ver. 1)

There are two ways of defining a React component:

- functional component: A function is a React component, iff
  - its return type is `React.Element`
  - the function has at least one parameter. This first parameter is considered as the `props` property. The type of the first parameter only has public fields (or methods defined by Object). The fields may be defined as optional.
- class component: A class is a React component, iff
  - it extends `React.Component`.
  - the type parameters must be structural types.
    - The first parameter defines the `props` type, it must only define public fields (or methods defined by Object)
    - The second parameter defines the `state` type, it must only define public fields (or methods defined by Object)

In both cases, the name must be capitalized (i.e., start with an upper case letter).

Component must be written capitalized (i.e. first letter upper case), html (or other xml elements) must be written lower case.

### JSX Basic Syntax

#### Requirement: IDE-241102: JSX Basic Syntax (ver. 1)

The following JSX syntax is supported (EBNF with model element assignments as used in Xtext). All other ECMAScript 2015 and N4JS syntax constructs are unaffected by this addition.

```
PrimaryExpression <Yield> returns n4js::Expression:  
... // as in N4JS  
| JSXElement  
;  
  
JSXElement:  
'<' jsxElementName=JSXElementName JSXAttributes  
(( '>' jsxChildren+=JSXChild* JSXClosingElement) | ('/' '>'));  
  
fragment JSXClosingElement *:  
'< '/' jsxClosingName=JSXElementName '>';  
  
JSXChild:  
JSXElement | JSXExpression  
;  
  
JSXExpression: '{' expression=AssignmentExpression<false,false> '}';  
  
JSXElementName:  
expression=JSXElementNameExpression  
;  
  
JSXElementNameExpression returns n4js::Expression:  
IdentifierRef<false>  
(n4js::ParameterizedPropertyAccessExpression target=current) ParameterizedPropertyAccessExpressionTail<false>)*  
;  
  
fragment JSXAttributes *:  
jsxAttributes+=JSXAttribute*;
```

```

JSXAttribute:
JSXSpreadAttribute
|
JSXPropertyAttribute;

JSXSpreadAttribute:
'{ ... }' expression=AssignmentExpression<false,false> '}'';

JSXPropertyAttribute:
property=types::IdentifiableElement/IdentifierName '=' (jsxAttributeValue=StringLiteral | ('{' jsxAttributeValue=AssignmentExpression<false,false> '}'));
```

This syntax is similar to the syntax defined by [JSX](#), except that

- JSXNamespaceName is not supported
- JSXText is not supported, instead, string template literals are to be used

Remarks on differences between the syntax defined in [\[Req-IDE-241102\]](#) and [JSX](#):

- JSXSelfClosingElement, JSXOpeningElement and JSXClosingElement are merged into one single rule JSXElement
- JSXAttributes is defined as fragment, that is, the attributes become fields of the JSXElement
- The different types of JSXElementName are defined by means of JSXElementNameExpression which is a simple expression, reusing the existing rules IdentifierRef and ParameterizedPropertyAccessExpression to model JSXIdentifier and JSXMemberExpression respectively
- JSXPropertyAttribute models the JSX non-spread JSXAttribute definition; again existing rules (IdentifierName, StringLiteral) are used to model JSX specific ones (JSXAttributeName, JSXDoubleStringCharacters, JSXSingleStringCharacters)

### JSX Syntax With Free Text

**Requirement: IDE-241401: JSX Syntax With Free Text** (ver. 1)

```

JSXChild:
JSXElement | JSXExpression | JSXText
;
```

### JSX Extended Syntax Check

**Requirement: IDE-241103: JSX Extended Syntax Check** (ver. 1)

\* It is an error, if corresponding opening and closing JSXElements have different names.

### JSX Expression Type

**Requirement: IDE-241113: JSX Expression Type** (ver. 1)

The type of a JSX expression is React.Element.

### React Symbol

**Requirement: IDE-241114: React Symbol** (ver. 1)

If a JSX literal is used, it is an error if the React symbol is not imported via

```
import React from "react"
```

### JSXElement names (tags)

**Requirement: IDE-241115: JSXElement names (tags)** (ver. 1)

\* It is an error if a capitalized tag cannot be bound to a function or class declaration.

### JSXElements referring to React components

**Requirement: IDE-241116: JSXElements referring to React components** (ver. 1)

\* It is an error, if a tag binds to a function declaration, which is not conform to the functional component definition. \* It is an error, if a tag binds to a class declaration, which is not conform to the class component definition.

#### JSXAttributes and React component properties

**Requirement: IDE-241117:** [JSXAttributes and React component properties](#) (ver. 1)

If the tag binds to a component, the following constraints must hold:

- The attribute must be a non-private field of the properties type.
- The tag should define attributes for all non-optional fields of the properties type. If no attribute is defined for a non-optional field, a warning is issued.
- The type of the attribute expression must be conform to the type of the corresponding `props`'s property

#### JSXElements referring to XML elements

**Requirement: IDE-241118:** [JSXElements referring to XML elements](#) (ver. 1)

If the lower-case tag does not bind to a function or class declaration, the following constraints must be hold:

- If the tag is neither a pre-defined HTML tag nor an SVG tag, a warning is issued.
- If an attribute of the tag is not a pre-defined property of the html tag or react specific attributes, a warning is issued. This requirement is currently NOT supported.

#### JSXSpreadAttribute behavior

**Requirement: IDE-241119:** [JSXSpreadAttribute behavior](#) (ver. 1)

The use of spread operators within an JSX element for specifying multiple attributes should be allowed. In this case, all constraints regarding type conformity checking and non-optional properties mentioned in [\[Req-IDE-241117\]](#) apply to the attributes specified in the spread operator. In particular,

- The type of each attribute specified in spread operator must be conform to the type of the corresponding property of `props`.
- If a non-optional property of `props` is specified neither as attribute nor in a spread operator, a warning is issued.

## 15.2. JSX Backend

The support for JSX in N4JS aims for an implementation that adheres to the idea of [React JSX](#). This means that JSX elements are transpiled to React Element factory calls (e.g. `<div prop="c">content</div>` transpiles to `React.createElement('div', {prop: "c"}, null)`). For that, the transpiler must be aware of a specific implementation of React and the corresponding `createElement` function.

#### React Implementation

**Requirement: GH-687:** [React Implementation](#) (ver. 1)

A react implementation is given in terms of a module that fulfils the following properties:

- The FQN of the module is `react`.
- Type definitions are available for the module.
- The module exports a function of name `createElement`.

If a react implementation is declared as project dependency, the N4JS transpiler automatically imports it to the module using JSX and generates the corresponding factory calls.



# Chapter 16. Grammars

N4JS extends the ECMAScript 2015 language grammar and combines it with type expression.



These grammars are slightly simplified versions of the "real" Xtext grammars used in the implementation. These grammars are post-processed and combined with additional validators so not all constructs are necessarily available in N4JS.

## 16.1. Type Expressions Grammar

### EBNF Type Expression Grammar

```
TypeRef:
    TypeRefWithoutModifiers => undefModifier=UndefModifierToken?
    | undefModifier=UndefModifierToken;

TypeRefWithoutModifiers:
    ((ParameterizedTypeRef | ThisTypeRef) => dynamic?='+'?)
    | ConstructorTypeRef
    | ClassifierTypeRef
    | FunctionTypeExpression
    | UnionTypeExpression
    | IntersectionTypeExpression;
;

TypeRefFunctionTypeExpression:
    ParameterizedTypeRef
    | ConstructorTypeRef
    | ClassifierTypeRef
    | UnionTypeExpression
    | IntersectionTypeExpression
    ;
;

TypeRefForCast:
    ParameterizedTypeRef
    | ThisTypeRef
    | ConstructorTypeRef
    | ClassifierTypeRef
    | FunctionTypeExpression;
;

TypeRefInClassifierType:
    ParameterizedTypeRefNominal
    | ThisTypeRefNominal;
;

ThisTypeRef:
    ThisTypeRefNominal | ThisTypeRefStructural;
;

ThisTypeRefNominal:
    'this';
;

ThisTypeRefStructural:
    definedTypingStrategy=TypingStrategyUseSiteOperator
    'this'
    ('with' TStructMemberList)?;
;

FunctionTypeExpression:
    '{'
    ('@' 'This' '(' declaredThisType=TypeRefFunctionTypeExpression ')')?
    'function'
    ('<' ownedTypeVars+=TypeVariable (',' ownedTypeVars+=TypeVariable)* '>')?
    ('(' TAnonymousFormalParameterList ')')
    ('returnTypeRef=TypeRef')?
    '}';
;

fragment TAnonymousFormalParameterList*:
    (fpars+=TAnonymousFormalParameter (',' fpars+=TAnonymousFormalParameter)*)?
    ;
;

TAnonymousFormalParameter:
    variadic?='...'? (=> name=TIdentifier ':')? typeRef=TypeRef
    ;
;

UnionTypeExpression:
```

```

'union' '{' typeRefs+=TypeRefWithoutModifiers (',' typeRefs+=TypeRefWithoutModifiers)* '}';
}

IntersectionTypeExpression:
  'intersection' '{' typeRefs+=TypeRefWithoutModifiers (',' typeRefs+=TypeRefWithoutModifiers)* '}';
}

ParameterizedTypeRef:
  ParameterizedTypeRefNominal | ParameterizedTypeRefStructural;

ParameterizedTypeRefStructural:
  definedTypingStrategy=TypingStrategyUseSiteOperator
  declaredType=[Type/TypeReferenceName]
  (=>'<' typeArgs+=TypeArgument (',' typeArgs+=TypeArgument)* '>')?
  ('with' TStructMemberList)?;

fragment TStructMemberList*: '{' (astStructuralMembers+=TStructMember (';' | ',')?)* '}';
}

TStructMember:
  TStructGetter
  | TStructSetter
  | TStructMethod
  | TStructField;

TStructMethod:
  =>
  (( '<' typeVars+=TypeVariable (',' typeVars+=TypeVariable)* '>')?
  name=TypesIdentifier '('
  ) TAnonymousFormalParameterList ')'
  (':' returnTypeRef=TypeRef)?;

;

TStructField:
  name=TypesIdentifier (':' returnTypeRef=TypeRef)?;
;

TStructGetter:
  => ('get'
  name=TypesIdentifier)
  ('(' ')' (':' declaredTypeRef=TypeRef)?);
;

TStructSetter:
  => ('set'
  name=TypesIdentifier)
  ('(' fpar=TAnonymousFormalParameter ')');
;

ParameterizedTypeRefNominal:
  declaredType=[Type/TypeReferenceName]
  (=> '<' typeArgs+=TypeArgument (',' typeArgs+=TypeArgument)* '>')?;

TypingStrategyUseSiteOperator:
  '~' ('~' | STRUCTMODSUFFIX)?;

TypingStrategyDefSiteOperator:
  '~';
;

terminal STRUCTMODSUFFIX:
  ('r' | 'i' | 'w') '~';
;

ConstructorTypeRef:
  'constructor' '{' staticTypeRef=TypeRefInClassifierType '}';
;

ClassifierTypeRef:
  'type' '{' staticTypeRef=TypeRefInClassifierType '}';
;

TypeReferenceName:
  IDENTIFIER ('.' IDENTIFIER)*;
;

TypeArgument:
  Wildcard | TypeRef;
;

Wildcard:
  => ('?') (( 'extends' declaredUpperBound=TypeRef) | ('super'
  declaredLowerBound=TypeRef))?;
;

UndefModifierToken:
  '?';
;

TypeVariable:

```

```

name=IDENTIFIER ('extends' declaredUpperBounds+=ParameterizedTypeRef ('&'  

declaredUpperBounds+=ParameterizedTypeRef)*)?;  
  

TypesIdentifier:  

  IDENTIFIER  

  | 'get' | 'set' | 'abstract' | 'project'  

  | 'union' | 'intersection'  

  | 'as' | 'from' | 'type' | 'void' | 'null';  
  

TIdentifier:  

  TypesIdentifier  

  | 'implements' | 'interface'  

  | 'private' | 'protected' | 'public'  

  | 'static'  

;  
  

terminal IDENTIFIER:  

  IDENTIFIER_START IDENTIFIER_PART*;  
  

terminal INT:  

  DECIMAL_INTEGER_LITERAL_FRAGMENT;  
  

terminal ML_COMMENT:  

  ML_COMMENT_FRAGMENT;  
  

terminal SL_COMMENT:  

  '//' (!LINE_TERMINATOR_FRAGMENT)*;  
  

terminal EOL:  

  LINE_TERMINATOR_SEQUENCE_FRAGMENT;  
  

terminal WS:  

  WHITESPACE_FRAGMENT*;  
  

terminal fragment UNICODE_ESCAPE_FRAGMENT:  

  '\\\\' ('u' (  

    HEX_DIGIT (HEX_DIGIT (HEX_DIGIT HEX_DIGIT?))?)?  

  | '{' HEX_DIGIT* '}'?)?  

)?);  
  

terminal fragment IDENTIFIER_START:  

  UNICODE_LETTER_FRAGMENT  

  | '$'  

  | '_'  

  | UNICODE_ESCAPE_FRAGMENT;  
  

terminal fragment IDENTIFIER_PART:  

  UNICODE_LETTER_FRAGMENT  

  | UNICODE_ESCAPE_FRAGMENT  

  | '$'  

  | UNICODE_COMBINING_MARK_FRAGMENT  

  | UNICODE_DIGIT_FRAGMENT  

  | UNICODE_CONNECTOR_PUNCTUATION_FRAGMENT  

  | ZWNJ  

  | ZWJ;  
  

terminal DOT_DOT:  

  '...'  

;

```

## 16.2. N4JS Language Grammar

```

Script: annotations+=ScriptAnnotation*  

scriptElements+=ScriptElement*;  
  

ScriptElement:  

  AnnotatedScriptElement  

  | N4ClassDeclaration<Yield=false>  

  | N4InterfaceDeclaration<Yield=false>  

  | N4EnumDeclaration<Yield=false>  

  | ImportDeclaration  

  | ExportDeclaration  

  | RootStatement<Yield=false>  

;  
  

AnnotatedScriptElement:  

  AnnotationList (  

    {ExportDeclaration.annotationList=current} ExportDeclarationImpl  

  | {ImportDeclaration.annotationList=current} ImportDeclarationImpl  

  | {FunctionDeclaration.annotationList=current}

```

```

=> ((declaredModifiers+=N4Modifier)* AsyncNoTrailingLineBreak
    ->FunctionImpl<Yield=false,YieldIfGenerator=false,Expression=false>)
|
|   (
|     (
|       {
|         {N4ClassDeclaration.annotationList=current}
|         (declaredModifiers+=N4Modifier)*
|         'class' typingStrategy=TypingStrategyDefSiteOperator?
|         name=BindingIdentifier<Yield=false>
|         TypeVariables?
|         ClassExtendsClause<Yield=false>?
|         | {N4InterfaceDeclaration.annotationList=current}
|         (declaredModifiers+=N4Modifier)*
|         'interface' typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield=false>
|         TypeVariables?
|         InterfaceImplementsList?
|       )
|       Members<Yield=false>
|     )
|   {N4EnumDeclaration.annotationList=current}
|   (declaredModifiers+=N4Modifier)*
|   'enum' name=BindingIdentifier<Yield=false>
|   '{'
|     literals+=N4EnumLiteral (',' literals+= N4EnumLiteral)*
|   '}'
|
;

fragment TypeVariables*:
'<' typeVars+=TypeVariable (',' typeVars+=TypeVariable)* '>'

;

ExportDeclaration:
ExportDeclarationImpl
;

fragment ExportDeclarationImpl*:
'export' (
  wildcardExport?='*' ExportFromClause Semi
| ExportClause ->ExportFromClause? Semi
| exportedElement=ExportableElement
| defaultExport?='default' (~>exportedElement=ExportableElement |
  defaultExportedExpression=AssignmentExpression<In=true,Yield=false> Semi)
)
;

fragment ExportFromClause*:
'from' reexportedFrom=[types::TModule/ModuleSpecifier]
;

fragment ExportClause*:
'{'
  (namedExports+=ExportSpecifier (',' namedExports+=ExportSpecifier)* ','?)?
'}'
;

ExportSpecifier:
element=IdentifierRef<Yield=false> ('as' alias=IdentifierName)?
;

ExportableElement:
AnnotatedExportableElement<Yield=false>
| N4ClassDeclaration<Yield=false>
| N4InterfaceDeclaration<Yield=false>
| N4EnumDeclaration<Yield=false>
| ExportedFunctionDeclaration<Yield=false>
| ExportedVariableStatement
;
;

AnnotatedExportableElement <Yield>:
AnnotationList (
  {FunctionDeclaration.annotationList=current}
  (declaredModifiers+=N4Modifier)* AsyncNoTrailingLineBreak
  FunctionImpl<Yield, Yield, Expression=false>
| {ExportedVariableStatement.annotationList=current}
  (declaredModifiers+=N4Modifier)*
  varStmtKeyword=VariableStatementKeyword
  varDeclsOrBindings+=ExportedVariableDeclarationOrBinding<Yield> (',' varDeclsOrBindings+=ExportedVariableDeclarationOrBinding<Yield> )* Semi
  |
  (
    (

```

```

{N4ClassDeclaration.annotationList=current}
(declaredModifiers+=N4Modifier)*
'class' typingStrategy=TypingStrategyDefSiteOperator?
name=BindingIdentifier<Yield>
TypeVariables?
ClassExtendsClause<Yield>?
| {N4InterfaceDeclaration.annotationList=current}
(declaredModifiers+=N4Modifier)*
('interface') typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield>
TypeVariables?
InterfaceImplementsList?
)
Members<Yield>
)
|
{N4EnumDeclaration.annotationList=current}
(declaredModifiers+=N4Modifier)*
'enum' name=BindingIdentifier<Yield>
{
literals+=N4EnumLiteral (',' literals+= N4EnumLiteral)*
}
)
;

ImportDeclaration:
ImportDeclarationImpl
;

fragment ImportDeclarationImpl*:
'import' (
ImportClause importFrom?= 'from'
)? module=[types::TModule/ModuleSpecifier] Semi
;

fragment ImportClause*:
importSpecifiers+=DefaultImportSpecifier (',' ImportSpecifiersExceptDefault)?
| ImportSpecifiersExceptDefault
;

fragment ImportSpecifiersExceptDefault*:
importSpecifiers+=NamespaceImportSpecifier
| '{' (importSpecifiers+=NamedImportSpecifier (',' importSpecifiers+=NamedImportSpecifier)* ','?)? '}'
;

NamedImportSpecifier:
importedElement=[types::TExportableElement/BindingIdentifier<Yield=false>]
| importedElement=[types::TExportableElement/IdentifierName] 'as' alias=BindingIdentifier<Yield=false>
;

DefaultImportSpecifier:
importedElement=[types::TExportableElement/BindingIdentifier<Yield=false>]
;

NamespaceImportSpecifier: '*' 'as' alias=BindingIdentifier<Yield=false> (declaredDynamic?= '+' )?;
ModuleSpecifier: STRING;

FunctionDeclaration <Yield>:
=> ((declaredModifiers+=N4Modifier)* AsyncNoTrailingLineBreak
-> FunctionImpl <Yield,Yield,Expression=false>
) => Semi?
;

fragment AsyncNoTrailingLineBreak *: (declaredAsync?= 'async' NoLineTerminator)?;

fragment FunctionImpl<Yield, YieldIfGenerator, Expression>*:
'function'
(
generator?= '*' FunctionHeader<YieldIfGenerator,Generator=true> FunctionBody<Yield=true,Expression>
| FunctionHeader<Yield,Generator=false> FunctionBody<Yield=false,Expression>
)
;

fragment FunctionHeader<Yield, Generator>*:
TypeVariables?
name=BindingIdentifier<Yield>?
StrictFormalParameters<Yield=Generator>
(-> !';' returnTypeRef=TypeRef)?
;

fragment FunctionBody <Yield, Expression>*:

```

```

<Expression> body=Block<Yield>
| <!Expression> body=Block<Yield>?
;

ExportedFunctionDeclaration<Yield>:
  FunctionDeclaration<Yield>
;

FunctionTypeExpression:
  {types::FunctionTypeExpression}
  '{'
  ('@' 'This' '(' declaredThisType=TypeRefFunctionTypeExpression ')')?
  'function'
  ('<' ownedTypeVars+=TypeVariable (',' ownedTypeVars+=TypeVariable)* '>')?
  '('
  (fpars+=TAnonymousFormalParameter (',' fpars+=TAnonymousFormalParameter)*)?
  ')'
  (':' returnTypeRef=TypeRef)?
  '}';

AnnotatedFunctionDeclaration <Yield, Default>:
  annotationList=AnnotationList
  (declaredModifiers+=N4Modifier)* AsyncNoTrailingLineBreak
  FunctionImpl<Yield, Yield, Expression=false>
;

FunctionExpression:
  (FunctionImpl<Yield=false, YieldIfGenerator=true, Expression=true>
  )
;

AsyncFunctionExpression:
  =>(declaredAsync?='async' NoLineTerminator 'function')
  FunctionHeader<Yield=false, Generator=false> FunctionBody<Yield=false, Expression=true>
;

ArrowExpression <In, Yield>:
  => (
    (
      ('(' (fpars+=FormalParameter<Yield>
        (',' fpars+=FormalParameter<Yield>)*)?
      ')') (':' returnTypeRef=TypeRef)?
      | =>(declaredAsync?='async' NoLineTerminator '(')
      (fpars+=FormalParameter<Yield> (',' fpars+=FormalParameter<Yield>)*)?
      ')') (':' returnTypeRef=TypeRef)?
      | fpars+=BindingIdentifierAsFormalParameter<Yield>
    )
    '=>'
  )
  (-> hasBracesAroundBody?='{ body=BlockMinusBraces<Yield> '}'
  | body=ExpressionDisguisedAsBlock<In>)
;
;

fragment StrictFormalParameters <Yield>*:
  ('(' (fpars+=FormalParameter<Yield> (',' fpars+=FormalParameter<Yield>)*)? ')')
;

BindingIdentifierAsFormalParameter <Yield>: name=BindingIdentifier<Yield>;
;

BlockMinusBraces <Yield>: statements+=Statement<Yield>*;
;

ExpressionDisguisedAsBlock <In>:
  statements+=AssignmentExpressionStatement<In>
;

AssignmentExpressionStatement <In>: expression=AssignmentExpression<In, Yield=false>;
;

AnnotatedExpression <Yield>:
  ExpressionAnnotationList (
    {N4ClassExpression.annotationList=current}
    'class' name=BindingIdentifier<Yield>?
    ClassExtendsClause<Yield>?
    Members<Yield>
    | {FunctionExpression.annotationList=current} AsyncNoTrailingLineBreak
    FunctionImpl<Yield=false, YieldIfGenerator=true, Expression=true>
  )
;
;

TypeVariable:

```

```

name=IdentifierOrThis
(   'extends' declaredUpperBounds+=ParameterizedTypeRefNominal
    ('&' declaredUpperBounds+=ParameterizedTypeRefNominal)*
) ?
;

FormalParameter <Yield>:
    BindingElementFragment<Yield>
;

fragment BindingElementFragment <Yield>*:
    (=> bindingPattern=BindingPattern<Yield>
    | annotations+=Annotation*
    (
        variadic?='...'? name=BindingIdentifier<Yield> ColonSepTypeRef?
    )
)
('=' initializer=AssignmentExpression<In=true, Yield>)?
;

fragment ColonSepTypeRef*:
    '::' declaredTypeRef=TypeRef
;

Block <Yield>: => ('{' statements+=Statement<Yield>* '}' );
RootStatement <Yield>:
    Block<Yield>
    | FunctionDeclaration<Yield>
    | VariableStatement<In=true, Yield>
    | EmptyStatement
    | LabelledStatement<Yield>
    | ExpressionStatement<Yield>
    | IfStatement<Yield>
    | IterationStatement<Yield>
    | ContinueStatement<Yield>
    | BreakStatement<Yield>
    | ReturnStatement<Yield>
    | WithStatement<Yield>
    | SwitchStatement<Yield>
    | ThrowStatement<Yield>
    | TryStatement<Yield>
    | DebuggerStatement
;
;

statement <Yield>:
    AnnotatedFunctionDeclaration<Yield, Default=false>
    | RootStatement<Yield>
;
;

enum VariableStatementKeyword:
    var='var' | const='const' | let='let'
;
;

VariableStatement <In, Yield>:
    =>(varStmtKeyword=VariableStatementKeyword
    )
    varDeclsOrBindings+=VariableDeclarationOrBinding<In, Yield, false>
    (',' varDeclsOrBindings+=VariableDeclarationOrBinding<In, Yield, false>)* Semi
;
;

ExportedVariableStatement:
    (declaredModifiers+=N4Modifier)*
    varStmtKeyword=VariableStatementKeyword
    varDeclsOrBindings+=ExportedVariableDeclarationOrBinding<Yield=false>
    (',' varDeclsOrBindings+=ExportedVariableDeclarationOrBinding<Yield=false>)* Semi
;
;

VariableDeclarationOrBinding <In, Yield, OptionalInit>:
    VariableBinding<In, Yield, OptionalInit>
    | VariableDeclaration<In, Yield, true>
;
;

VariableBinding <In, Yield, OptionalInit>:
    => pattern=BindingPattern<Yield> (
        <OptionalInit> ('=' expression=AssignmentExpression<In, Yield>)?
        | <!OptionalInit> '=' expression=AssignmentExpression<In, Yield>
    )
;
;

VariableDeclaration <In, Yield, AllowType>:

```

```

VariableDeclarationImpl<In,Yield,AllowType>;
;

fragment VariableDeclarationImpl <In, Yield, AllowType>*:
    annotations+=Annotation*
    (
        <AllowType> =>(
            name=BindingIdentifier<Yield> ColonSepTypeRef?
            ) ('=' expression=AssignmentExpression<In,Yield>)?
        | <!AllowType> =>(
            name=BindingIdentifier<Yield>
            ) ('=' expression=AssignmentExpression<In,Yield>)?
        )
    )
;

ExportedVariableDeclarationOrBinding <Yield>:
    ExportedVariableBinding<Yield>
    | ExportedVariableDeclaration<Yield>
;
;

ExportedVariableBinding <Yield>:
    => pattern=BindingPattern<Yield> '=' expression=AssignmentExpression<In=true,Yield>
;

ExportedVariableDeclaration <Yield>:
    VariableDeclarationImpl<In=true,Yield,AllowType=true>
;
EmptyStatement: '';
ExpressionStatement <Yield>: expression=Expression<In=true,Yield> Semi;
;

IfStatement <Yield>: 'if' '(' expression=Expression<In=true,Yield> ')'
    ifStmt=Statement<Yield> (=> 'else' elseStmt=Statement<Yield>)?;
;

IterationStatement <Yield>:
    DoStatement<Yield>
    | WhileStatement<Yield>
    | ForStatement<Yield>
;
;

DoStatement <Yield>: 'do' statement=Statement<Yield> 'while'
    '(' expression=Expression<In=true,Yield> ')' => Semi?;
WhileStatement <Yield>: 'while' '(' expression=Expression<In=true,Yield> ')'
    statement=Statement<Yield>;
;

ForStatement <Yield>:
    'for' '('
    (
        =>(initExpr=LetIdentifierRef forIn?='in' expression=Expression<In=true,Yield> ')')
    | ( ->varStmtKeyword=VariableStatementKeyword
        (
            =>(varDeclsOrBindings+=BindingIdentifierAsVariableDeclaration<In=false,Yield>
                (forIn?='in' | forOf?='of') ->expression=AssignmentExpression<In=true,Yield>?)
            | varDeclsOrBindings+=VariableDeclarationOrBinding<In=false,Yield,OptionalInit=true>
                (
                    (';' varDeclsOrBindings+=VariableDeclarationOrBinding<In=false,Yield,false>)* ';'
                    expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
                    | forIn?='in' expression=Expression<In=true,Yield>?
                    | forOf?='of' expression=AssignmentExpression<In=true,Yield>?
                )
            )
        | initExpr=Expression<In=false,Yield>
        (
            ';' expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
            | forIn?='in' expression=Expression<In=true,Yield>?
            | forOf?='of' expression=AssignmentExpression<In=true,Yield>?
        )
        |
        '|'; expression=Expression<In=true,Yield>? ';' updateExpr=Expression<In=true,Yield>?
    )
    )
;
) statement=Statement<Yield>;
;

LetIdentifierRef:
    id={types::IdentifiableElement|LetAsIdentifier}
;
;

LetAsIdentifier: 'let';
;

BindingIdentifierAsVariableDeclaration <In, Yield>:
    name=BindingIdentifier<Yield>
;
;
```

```

ContinueStatement <Yield>: 'continue' (label=[LabelledStatement/BindingIdentifier<Yield>])? Semi;

BreakStatement <Yield>: 'break' (label=[LabelledStatement/BindingIdentifier<Yield>])? Semi;

ReturnStatement <Yield>: 'return' (expression=Expression<In=true,Yield>)? Semi;

WithStatement <Yield>: 'with' '(' expression=Expression<In=true,Yield> ')' statement=Statement<Yield>;

SwitchStatement <Yield>:
    'switch' '(' expression=Expression<In=true,Yield> ')' '{'
    (cases+=CaseClause<Yield>)*
    ((cases+=DefaultClause<Yield>)
     (cases+=CaseClause<Yield>)*)? '}'
;

CaseClause <Yield>: 'case' expression=Expression<In=true,Yield> ':' (statements+=Statement<Yield>)*;

DefaultClause <Yield>: 'default' ':' (statements+=Statement<Yield>)*;

LabelledStatement <Yield>: => (name=BindingIdentifier<Yield> ':') statement=Statement<Yield>;

ThrowStatement <Yield>:
    'throw' expression=Expression<In=true,Yield> Semi;

TryStatement <Yield>:
    'try' block=Block<Yield>
    ((catch=CatchBlock<Yield> finally=FinallyBlock<Yield>?) | finally=FinallyBlock<Yield>)
;
CatchBlock <Yield>: 'catch' '(' catchVariable=CatchVariable<Yield> ')' block=Block<Yield>;

CatchVariable <Yield>:
    =>bindingPattern=BindingPattern<Yield>
    | =>(name=BindingIdentifier<Yield> -> ColonSepTypeRef)
    | name=BindingIdentifier<Yield>
;
FinallyBlock <Yield>: 'finally' block=Block<Yield>;

DebuggerStatement:
    'debugger' Semi;

PrimaryExpression <Yield>:
    ThisLiteral
    | SuperLiteral
    | IdentifierRef<Yield>
    | ParameterizedCallExpression<Yield>
    | Literal
    | ArrayLiteral<Yield>
    | ObjectLiteral<Yield>
    | ParenExpression<Yield>
    | AnnotatedExpression<Yield>
    | FunctionExpression
    | AsyncFunctionExpression
    | N4ClassExpression<Yield>
    | TemplateLiteral<Yield>
;
ParenExpression <Yield>: '(' expression=Expression<In=true,Yield> ')';

IdentifierRef <Yield>:
    id=[types.:IdentifiableElement/BindingIdentifier<Yield>]
;

superLiteral: 'super';

ThisLiteral: 'this';

ArrayLiteral <Yield>:
    '['
    elements+=ArrayPadding* (
        elements+=ArrayElement<Yield>
        (',' elements+=ArrayPadding* elements+=ArrayElement<Yield>)*
        (trailingComma?=',' elements+=ArrayPadding*)?
    )?
    ']'
;
ArrayPadding: ',' ;

```

```

ArrayElement <Yield>: spread?='...' expression=AssignmentExpression<In=true,Yield>;

ObjectLiteral <Yield>: '{'
  ( propertyAssignments+=PropertyAssignment<Yield>
    ( ',' propertyAssignments+=PropertyAssignment<Yield>)* ','?
  )?
}
;

PropertyAssignment <Yield>:
  AnnotatedPropertyAssignment<Yield>
  | PropertyValuePair<Yield>
  | PropertyGetterDeclaration<Yield>
  | PropertySetterDeclaration<Yield>
  | PropertyMethodDeclaration<Yield>
  | PropertyValuePairSingleName<Yield>
;
;

AnnotatedPropertyAssignment <Yield>:
  PropertyAssignmentAnnotationList (
    => ( {PropertyNameValuePair.annotationList=current} declaredTypeRef=TypeRef?
      LiteralOrComputedPropertyName<Yield> ':'
    ) expression=AssignmentExpression<In=true,Yield>
  | => ( {PropertyGetterDeclaration.annotationList=current}
    GetterHeader<Yield>
    ) body=Block<Yield=false>
  | => ( {PropertySetterDeclaration.annotationList=current}
    'set' ->LiteralOrComputedPropertyName <Yield>
    ) ('(' fpar=FormalParameter<Yield> ')' body=Block<Yield=false>
  | => ( {PropertyMethodDeclaration.annotationList=current}
    TypeVariables? returnTypeRef=TypeRef?
    (generator?='*' LiteralOrComputedPropertyName<Yield> ->MethodParamsAndBody <Generator=true>
      | LiteralOrComputedPropertyName<Yield> -> MethodParamsAndBody <Generator=false>
    )
    ) ';'?
  | {PropertyNameValuePairSingleName.annotationList=current}
    declaredTypeRef=TypeRef? identifierRef=IdentifierRef<Yield>
    ( '=' expression=AssignmentExpression<In=true,Yield>)?)
);
;

PropertyMethodDeclaration <Yield>:
  => ( TypeVariables? returnTypeRef=TypeRef?
    (
      generator?='*' LiteralOrComputedPropertyName<Yield>
      ->MethodParamsAndBody<Generator=true>
      | LiteralOrComputedPropertyName<Yield> ->MethodParamsAndBody <Generator=false>
    )
  )
  ';'?
;
;

PropertyNameValuePair <Yield>:
  => (
    declaredTypeRef=TypeRef? LiteralOrComputedPropertyName<Yield> ':'
  )
  expression=AssignmentExpression<In=true,Yield>
;
;

PropertyNameValuePairSingleName <Yield>:
  declaredTypeRef=TypeRef?
  identifierRef=IdentifierRef<Yield>
  ( '=' expression=AssignmentExpression<In=true,Yield>)?
;
;

PropertyGetterDeclaration <Yield>:
  => (
    GetterHeader<Yield>
  )
  body=Block<Yield=false>
;
;

PropertySetterDeclaration <Yield>:
  => (
    'set'
    ->LiteralOrComputedPropertyName <Yield>
  )
  ('(' fpar=FormalParameter<Yield> ')') body=Block<Yield=false>
;
;
```

```

ParameterizedCallExpression <Yield>:
  TypeArguments
  target=IdentifierRef<Yield>
  ArgumentsWithParentheses<Yield>
;

LeftHandSideExpression <Yield>:
  MemberExpression<Yield> (
    {ParameterizedCallExpression.target=current} ArgumentsWithParentheses<Yield>
    (
      {ParameterizedCallExpression.target=current} ArgumentsWithParentheses<Yield>
      | {IndexedAccessExpression.target=current} IndexedAccessExpressionTail<Yield>
      | {ParameterizedPropertyAccessExpression.target=current}
        ParameterizedPropertyAccessExpressionTail<Yield>
      | -> ({TaggedTemplateString.target=current} template=TemplateLiteral<Yield>)
    )*
  )?
;

fragment Arguments <Yield>*:
  arguments+=AssignmentExpression<In=true,Yield>
  (', ' arguments+=AssignmentExpression<In=true,Yield>)*
  ('...' spread?=...' arguments+=AssignmentExpression<In=true,Yield>)?
  | spread?='...' arguments+=AssignmentExpression<In=true,Yield>
;

fragment TypeArguments*:
  '<' typeArgs+=TypeRef (',' typeArgs+=TypeRef)* '>'
;

fragment ArgumentsWithParentheses <Yield>*:
  '(' Arguments<Yield>? ')'
;

MemberExpression <Yield>:
  =>('new' '.') 'target'
  | => ('new') callee=MemberExpression<Yield> (-> TypeArguments)?
  (=> withArgs?='(' Arguments<Yield>? ')'
    (
      {IndexedAccessExpression.target=current} IndexedAccessExpressionTail<Yield>
      | {ParameterizedPropertyAccessExpression.target=current}
        ParameterizedPropertyAccessExpressionTail<Yield>
      | {TaggedTemplateString.target=current} template=TemplateLiteral<Yield>
    )*
  )?
  | PrimaryExpression<Yield> (
    {IndexedAccessExpression.target=current} IndexedAccessExpressionTail<Yield>
    | {ParameterizedPropertyAccessExpression.target=current}
      ParameterizedPropertyAccessExpressionTail<Yield>
    | {TaggedTemplateString.target=current} template=TemplateLiteral<Yield>
  )*
;

fragment IndexedAccessExpressionTail <Yield>*:
  '[' index=Expression<In=true,Yield> ']'
;

fragment ParameterizedPropertyAccessExpressionTail <Yield>*:
  '..' TypeArguments? property=[types::IdentifiableElement/IdentifierName]
;

PostfixExpression <Yield>:
  LeftHandSideExpression<Yield> (
    =>({PostfixExpression.expression=current} op=PostfixOperator
    )
  )?
;

enum PostfixOperator: inc='++' | dec='--';

CastExpression <Yield>: PostfixExpression<Yield>
  (=>({CastExpression.expression=current} 'as') targetTypeRef=TypeRefForCast)?

UnaryExpression <Yield>:
  CastExpression<Yield>
  | (op=UnaryOperator expression=UnaryExpression<Yield>);

enum UnaryOperator: delete | void | typeof | inc='++' | dec='--' | pos='+' | neg='-' | inv='~' | not='!';

MultiplicativeExpression <Yield>: UnaryExpression<Yield>

```

```

(=> ({MultiplicativeExpression.lhs=current} op=MultiplicativeOperator)
    rhs=UnaryExpression<Yield>)*;

enum MultiplicativeOperator: times='*' | div='/` | mod='%;

AdditiveExpression <Yield>: MultiplicativeExpression<Yield>
(=> ({AdditiveExpression.lhs=current} op=AdditiveOperator)
    rhs=MultiplicativeExpression<Yield>)*;

enum AdditiveOperator: add='+' | sub='-';

ShiftExpression <Yield>: AdditiveExpression<Yield>
(=> ({ShiftExpression.lhs=current} op=ShiftOperator rhs=AdditiveExpression<Yield>))*;
;

ShiftOperator:
  '>' | '>' | '>'?
  | '<<'?
;

RelationalExpression <In, Yield>: ShiftExpression<Yield>
(=> ({RelationalExpression.lhs=current} op=RelationalOperator<In>
    ->rhs=ShiftExpression<Yield>)*;

RelationalOperator <In>:
  '<' | '>' | '<=' | '>=' | 'instanceof' | <In> 'in';

EqualityExpression <In, Yield>: RelationalExpression<In,Yield>
(=> ({EqualityExpression.lhs=current} op=EqualityOperator) rhs=RelationalExpression<In,Yield>)*;

enum EqualityOperator: same='===' | nsame='!==' | eq='==' | neq='!=';

BitwiseANDExpression <In, Yield>: EqualityExpression<In,Yield>
(=> ({BinaryBitwiseExpression.lhs=current} op=BitwiseANDOperator) rhs=EqualityExpression<In,Yield>)*;

BitwiseANDOperator: '&';

BitwiseXORExpression <In, Yield>: BitwiseANDExpression<In,Yield>
(=> ({BinaryBitwiseExpression.lhs=current} op=BitwiseXOROperator) rhs=BitwiseANDExpression<In,Yield>)*;

BitwiseXOROperator: '^';

BitwiseORExpression <In, Yield>: BitwiseXORExpression<In,Yield>
(=> ({BinaryBitwiseExpression.lhs=current} op=BitwiseOROperator) rhs=BitwiseXORExpression<In,Yield>)*;

BitwiseOROperator: '|';

LogicalANDExpression <In, Yield>: BitwiseORExpression<In,Yield>
(=> ({BinaryLogicalExpression.lhs=current} op=LogicalANDOperator) rhs=BitwiseORExpression<In,Yield>)*;

LogicalANDOperator: '&&';

LogicalORExpression <In, Yield>: LogicalANDExpression<In,Yield>
(=> ({BinaryLogicalExpression.lhs=current} op=LogicalOROperator) rhs=LogicalANDExpression<In,Yield>)*;

LogicalOROperator: '||';

ConditionalExpression <In, Yield>: LogicalORExpression<In,Yield>
(=> ({ConditionalExpression.expression=current} '?') trueExpression=AssignmentExpression<In=true,Yield>
    ':' falseExpression=AssignmentExpression<In,Yield>)?;

AssignmentExpression <In, Yield>:
  AwaitExpression<In,Yield>
  | PromisifyExpression<In,Yield>
  | ArrowExpression<In,Yield>
  | <Yield> YieldExpression<In>
  | ConditionalExpression<In,Yield>
    (=> ({AssignmentExpression.lhs=current} op=AssignmentOperator)
      rhs=AssignmentExpression<In,Yield>)?
;

YieldExpression <In>:
  'yield' => many?='*'? -> expression=AssignmentExpression<In,Yield=true>?
;

AssignmentOperator:
  '=' | '*=' | '/=' | '%=' | '+=' | '-='
  | '<<='
  | '>' | '>'? | '>='
  | '&=' | '^=' | '|='
;
```

```

;

AwaitExpression <In, Yield>:
  => ('await') expression=AssignmentExpression<In,Yield>;

PromisifyExpression <In, Yield>:
  => ('@' 'Promisify') expression=AssignmentExpression<In,Yield>;

Expression <In, Yield>:
  AssignmentExpression<In,Yield> ({CommaExpression.exprs+=current}
  ',' exprs+=AssignmentExpression<In,Yield>
  (',' exprs+=AssignmentExpression<In,Yield>)*)?
;

TemplateLiteral <Yield>:
  (
    segments+=NoSubstitutionTemplate
    | segments+=TemplateHead segments+=Expression<In=true,Yield>? TemplateExpressionEnd
      (
        segments+=TemplateMiddle segments+=Expression<In=true,Yield>?
        TemplateExpressionEnd
      )*
    segments+=TemplateTail
  )
;

TemplateExpressionEnd:
  '}'
;

NoSubstitutionTemplate:
  rawValue=NO_SUBSTITUTION_TEMPLATE_LITERAL
;

TemplateHead:
  rawValue=TEMPLATE_HEAD
;

TemplateTail:
  rawValue=TemplateTailLiteral;
;

TemplateMiddle:
  rawValue=TemplateMiddleLiteral;
;

Literal:
  NumericLiteral | BooleanLiteral | StringLiteral
  | NullLiteral | RegularExpressionLiteral;
NullLiteral: 'null';
BooleanLiteral: (true?=true' | 'false');
StringLiteral: value=STRING;
NumericLiteral:
  DoubleLiteral | IntLiteral | BinaryIntLiteral | OctalIntLiteral
  | LegacyOctalIntLiteral | HexIntLiteral | ScientificIntLiteral;
DoubleLiteral: value=DOUBLE;
IntLiteral: value=INT;
OctalIntLiteral: value=OCTAL_INT;
LegacyOctalIntLiteral: value=LEGACY_OCTAL_INT;
HexIntLiteral: value=HEX_INT;
BinaryIntLiteral: value=BINARY_INT;
ScientificIntLiteral: value=SCIENTIFIC_INT;
RegularExpressionLiteral: value=REGEX_LITERAL;

NumericLiteralAsString:
  DOUBLE | INT | OCTAL_INT | HEX_INT | SCIENTIFIC_INT
;

IdentifierOrThis:
  IDENTIFIER
  | 'This'
  | 'Promisify'
  | 'target';
;

AnnotationName:
  IDENTIFIER
  | 'This'
  | 'target';
;

BindingIdentifier <Yield>:
  IDENTIFIER
  | <!Yield> 'yield'
;
```

```

| N4Keyword
;

IdentifierName:
  IDENTIFIER | ReservedWord | N4Keyword
;

ReservedWord:
  'break' | 'case' | 'catch' | 'class' | 'const' | 'continue' | 'debugger' | 'default' | 'delete'
  | 'do' | 'else' | 'export' | 'extends' | 'finally' | 'for' | 'function' | 'if' | 'import'
  | 'in' | 'instanceof' | 'new' | 'return' | 'super' | 'switch' | 'this' | 'throw' | 'try'
  | 'typeof' | 'var' | 'void' | 'while' | 'with' | 'yield'
  | 'null'
  | 'true' | 'false'
  | 'enum';
;

N4Keyword:
  'get' | 'set'
  | 'let'
  | 'project'
  | 'external' | 'abstract' | 'static'
  | 'as' | 'from' | 'constructor' | 'of' | 'target'
  | 'type' | 'union' | 'intersection'
  | 'This' | 'Await' | 'Promisify'
  | 'await'
  | 'async'
  | 'implements' | 'interface'
  | 'private' | 'protected' | 'public'
;

SymbolLiteralComputedName <Yield>:
  BindingIdentifier<Yield> ('.' IdentifierName)?
;

terminal DOUBLE:
  '.' DECIMAL_DIGIT_FRAGMENT+ EXPONENT_PART?
  | DECIMAL_INTEGER_LITERAL_FRAGMENT '.' DECIMAL_DIGIT_FRAGMENT* EXPONENT_PART?
;
;

terminal HEX_INT: '0' ('x' | 'X') INT_SUFFIX;
;

terminal BINARY_INT: '0' ('b' | 'B') INT_SUFFIX;
;

terminal OCTAL_INT: '0' ('o' | 'O') INT_SUFFIX;
;

terminal LEGACY_OCTAL_INT: '0' DECIMAL_DIGIT_FRAGMENT INT_SUFFIX;
;

terminal fragment INT_SUFFIX: IDENTIFIER_PART*;
;

terminal SCIENTIFIC_INT:
  DECIMAL_INTEGER_LITERAL_FRAGMENT EXPONENT_PART
;
;

terminal fragment EXPONENT_PART:
  ('e' | 'E') SIGNED_INT
  | IDENTIFIER
;
;

terminal fragment SIGNED_INT:
  ('+' | '-') DECIMAL_DIGIT_FRAGMENT+ IDENTIFIER?
;
;

terminal STRING:
  """ DOUBLE_STRING_CHAR* """
  | ''' SINGLE_STRING_CHAR* '''
;
;

terminal fragment DOUBLE_STRING_CHAR:
  !(LINE_TERMINATOR_FRAGMENT | '\n' | '\r')
  | '\\"' (LINE_TERMINATOR_SEQUENCE_FRAGMENT | !LINE_TERMINATOR_FRAGMENT)?
;
;

terminal fragment SINGLE_STRING_CHAR:
  !(LINE_TERMINATOR_FRAGMENT | '\"' | '\\\\')
  | '\\\\' (LINE_TERMINATOR_SEQUENCE_FRAGMENT | !LINE_TERMINATOR_FRAGMENT)?
;
;

terminal fragment BACKSLASH_SEQUENCE:
  '\\\\' !(LINE_TERMINATOR_FRAGMENT)?
;
;
```

```

terminal fragment REGEX_CHAR:
    !(LINE_TERMINATOR_FRAGMENT | '\n' | '\r' | '[')
    | BACKSLASH_SEQUENCE
    | '[' REGEX_CHAR_OR_BRACKET* ']'??
;

terminal fragment REGEX_CHAR_OR_BRACKET:
    !(LINE_TERMINATOR_FRAGMENT | '\n' | ']')
    | BACKSLASH_SEQUENCE
;

REGEX_LITERAL:
    ('/' | '/=') REGEX_TAIL?
;

terminal fragment ACTUAL_REGEX_TAIL:
    REGEX_CHAR+ ('/' IDENTIFIER_PART*)?
    | '/' IDENTIFIER_PART*
;

terminal fragment REGEX_START:
    ('/' | '/=')
;

terminal REGEX_TAIL: // post processed
    '/1'
;
terminal TEMPLATE_HEAD:
    "``" TEMPLATE_LITERAL_CHAR* '$'+ '{'
;
terminal NO_SUBSTITUTION_TEMPLATE_LITERAL:
    ```" TEMPLATE_LITERAL_CHAR* '$'* ```"??
;
terminal fragment ACTUAL_TEMPLATE_END:
    TEMPLATE_LITERAL_CHAR* ('$'+ ('{' | '^'? | '^'?) | '^'?)
;
terminal fragment TEMPLATE_LITERAL_CHAR:
    !(LINE_TERMINATOR_FRAGMENT | '^' | '\n' | '$')
    | '$'+ !'{'
    | LINE_TERMINATOR_SEQUENCE_FRAGMENT
    | '\n' (LINE_TERMINATOR_SEQUENCE_FRAGMENT | !LINE_TERMINATOR_FRAGMENT)?
;
TemplateTailLiteral:
    TEMPLATE_END?
;
TemplateMiddleLiteral:
    TEMPLATE_MIDDLE
;
terminal TEMPLATE_MIDDLE:
    '/2' // will never be lexed
;
terminal TEMPLATE_END:
    '/3' // will never be lexed
;
terminal fragment TEMPLATE_CONTINUATION:
    '/4' // actually '}'
;
Semi: ';' // automatic semicolon insertion, post-processed
fragment NoLineTerminator*: NO_LINE_TERMINATOR?;

terminal NO_LINE_TERMINATOR:
    '/5' // post-processed, will never be lexed
;
Annotation: '@' AnnotationNoAtSign;
ScriptAnnotation: '@@' AnnotationNoAtSign;

AnnotationNoAtSign:
    name=AnnotationName (=> '(' args+=AnnotationArgument (',' args+=AnnotationArgument)* ')')?;

```

```

AnnotationArgument:
    LiteralAnnotationArgument | TypeRefAnnotationArgument
;

LiteralAnnotationArgument:
    literal=Literal
;

TypeRefAnnotationArgument:
    typeRef=TypeRef
;

AnnotationList:
    =>('@' -> annotations+=AnnotationNoAtSign) annotations+=Annotation*
;

ExpressionAnnotationList:
    annotations+=Annotation+
;

PropertyAssignmentAnnotationList:
    annotations+=Annotation+
;

N4MemberAnnotationList:
    {N4MemberAnnotationList} annotations+=Annotation+
;

TypeReferenceName:
    'void' | 'This' | 'await' | 'Promisify' | 'target' | QualifiedTypeReferenceName
;

QualifiedTypeReferenceName:
    IDENTIFIER ('.' IDENTIFIER)?
;

N4ClassDeclaration <Yield>:
    =>(
        {N4ClassDeclaration}
        (declaredModifiers+=N4Modifier)*
        'class' typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield>?
    )
    TypeVariables?
    ClassExtendsClause<Yield>?
    Members<Yield>
;

fragment Members <Yield>*:
    '{'
    ownedMembersRaw+=N4MemberDeclaration<Yield>*
    '}'
;

fragment ClassExtendsClause <Yield>*:
    'extends' (
        =>superClassRef=ParameterizedTypeRefNominal ('implements' ClassImplementsList)?
        | superClassExpression=LeftHandSideExpression<Yield>
    )
    | 'implements' ClassImplementsList
;

fragment ClassImplementsList*:
    implementedInterfaceRefs+=ParameterizedTypeRefNominal
    (',' implementedInterfaceRefs+=ParameterizedTypeRefNominal)*
;

N4ClassExpression <Yield>:
    {N4ClassExpression}
    'class' name=BindingIdentifier<Yield>?
    ClassExtendsClause<Yield>?
    Members<Yield>;
N4InterfaceDeclaration <Yield>:
    => (
        {N4InterfaceDeclaration}
        (declaredModifiers+=N4Modifier)*
        'interface' typingStrategy=TypingStrategyDefSiteOperator? name=BindingIdentifier<Yield>?
    )
    TypeVariables?
    InterfaceImplementsList?
    Members<Yield>
;

```

```

fragment InterfaceImplementsList*:
    'extends' superInterfaceRefs+=ParameterizedTypeRefNominal
        (',' superInterfaceRefs+=ParameterizedTypeRefNominal)*
;
N4EnumDeclaration <Yield>:
    => (
        {N4EnumDeclaration}
        (declaredModifiers+=N4Modifier)*
        'enum' name=BindingIdentifier<Yield>?
    )
    '{'
        (literals+=N4EnumLiteral (',' literals+=N4EnumLiteral))*?
    '}'
;
N4EnumLiteral: name=IdentifierOrThis ('::' value=STRING)?;

enum N4Modifier: // validator applies further checks
    private | project | protected | public
    | external | abstract | static | const;

N4MemberDeclaration <Yield>:
    AnnotatedN4MemberDeclaration<Yield>
    | N4GetterDeclaration<Yield>
    | N4SetterDeclaration<Yield>
    | N4MethodDeclaration<Yield>
    | N4FieldDeclaration<Yield>
    | N4CallableConstructorDeclaration<Yield>
;
AnnotatedN4MemberDeclaration <Yield> returns N4MemberDeclaration:
    N4MemberAnnotationList (
        => ({N4GetterDeclaration.annotationList=current}
            (declaredModifiers+=N4Modifier)* GetterHeader<Yield>) (body=Block<Yield>)? ';'?
        | => ({N4SetterDeclaration.annotationList=current}
            (declaredModifiers+=N4Modifier)* 'set' -> LiteralOrComputedPropertyName <Yield>)
            ('(' fpar=FormalParameter<Yield> ')') (body=Block<Yield>)? ';'?
        | => (
            {N4MethodDeclaration.annotationList=current} (declaredModifiers+=N4Modifier)* TypeVariables?
            (
                generator?='*' LiteralOrComputedPropertyName<Yield>
                ->MethodParamsReturnAndBody <Generator=true>
                | AsyncNoTrailingLineBreak LiteralOrComputedPropertyName<Yield>
                ->MethodParamsReturnAndBody <Generator=false>
            )
            ')';'?
        | {N4FieldDeclaration.annotationList=current} FieldDeclarationImpl<Yield>
        )
    )
;
fragment LiteralOrComputedPropertyName <Yield>*:
    name=IdentifierName | name=STRING | name=NumericLiteralAsString
    | '[' (=> ((name=SymbolLiteralComputedName<Yield> | name=StringLiteralAsName) ']')
        | computeNameFrom=AssignmentExpression<In=true,Yield> ']')
;
fragment LiteralPropertyName <Yield>*:
    name=IdentifierName | name=STRING | name=NumericLiteralAsString
    | '[' (name=SymbolLiteralComputedName<Yield> | name=StringLiteralAsName) ']'
;
stringLiteralAsName:
    STRING
;
fragment FieldDeclarationImpl <Yield>*:
    (declaredModifiers+=N4Modifier)*
    LiteralPropertyName<Yield> ColonSepTypeRef? ('=' expression=Expression<In=true,Yield>)? ';'*
;
N4FieldDeclaration <Yield>:
    {N4FieldDeclaration}
    FieldDeclarationImpl<Yield>
;
N4MethodDeclaration <Yield>:
    => ({N4MethodDeclaration} (declaredModifiers+=N4Modifier)* TypeVariables?
        (
            generator?='*' LiteralOrComputedPropertyName<Yield>
            ->MethodParamsReturnAndBody <Generator=true>

```

```

        | AsyncNoTrailingLineBreak LiteralOrComputedPropertyName<Yield>
        ->MethodParamsReturnAndBody <Generator=false>
    )
)
) ';'?
;

N4CallableConstructorDeclaration <Yield> returns N4MethodDeclaration:
    MethodParamsReturnAndBody <Generator=false> ';'?
;

fragment MethodParamsAndBody <Generator>*:
    StrictFormalParameters<Yield=Generator>
    (body=Block<Yield=Generator>)?
;

fragment MethodParamsReturnAndBody <Generator>*:
    StrictFormalParameters<Yield=Generator>
    ( ';' returnTypeRef=TypeRef )?
    (body=Block<Yield=Generator>)?
;

N4GetterDeclaration <Yield>:
    => ( {N4GetterDeclaration}
    (declaredModifiers+=N4Modifier)*
    GetterHeader<Yield>
    (body=Block<Yield>)? ';'?
;

fragment GetterHeader <Yield>*:
    ('get' -> LiteralOrComputedPropertyName <Yield> '(' ')') ColonSepTypeRef?
;

N4SetterDeclaration <Yield>:
    =>( {N4SetterDeclaration}
    (declaredModifiers+=N4Modifier)*
    'set'
    ->LiteralOrComputedPropertyName <Yield>
    )
    ('(' fpar=FormalParameter<Yield> ')') (body=Block<Yield>)? ';'?
;

BindingPattern <Yield>:
    ObjectBindingPattern<Yield>
    | ArrayBindingPattern<Yield>
;

ObjectBindingPattern <Yield>:
    '{' (properties+=BindingProperty<Yield,AllowType=false>
        (',' properties+=BindingProperty<Yield,AllowType=false>)* )? '}'
;

ArrayBindingPattern <Yield>:
    '['
        elements+=Elision* (
            elements+=BindingRestElement<Yield>
            (',' elements+=Elision* elements+=BindingRestElement<Yield>)*
            (',' elements+=Elision*)?
        )?
    ']'
;

BindingProperty <Yield, AllowType>:
    =>(LiteralBindingPropertyName<Yield> ':') value=BindingElement<Yield>
    | value=SingleNameBinding<Yield,AllowType>
;

fragment LiteralBindingPropertyName <Yield>*:
    declaredName=IdentifierName | declaredName=STRING | declaredName=NumericLiteralAsString
    | '[' (declaredName=SymbolLiteralComputedName<Yield> | declaredName=STRING) ']'
;

SingleNameBinding <Yield, AllowType>:
    varDecl=VariableDeclaration<In=true,Yield,AllowType>
;

BindingElement <Yield>:
    =>(nestedPattern=BindingPattern<Yield>) ('=' expression=AssignmentExpression<In=true,Yield>)?
    | varDecl=VariableDeclaration<In=true,Yield,AllowType=true>
;

```

```
BindingRestElement <Yield>:
  rest?='...'?
  (
    =>(nestedPattern=BindingPattern<Yield>)
    ('=' expression=AssignmentExpression<In=true,Yield>)?
    | varDecl=VariableDeclaration<In=true,Yield,AllowType=true>
  )
;

Elision:
  '_'
;
```



# Chapter 17. JSObjects

The built-in ECMAScript Objects [ECMA11a(p.S15, p.p.102)] are supported and their properties are annotated with types as described in this chapter. The semantics of these properties do not change. The short description is copied from [ECMA11a] repeated here for convenience.

## 17.1. Object

`Object` is the super type of all declared types and `N4Object`. It is almost similar to the JavaScript type `object` except that no properties may be dynamically added to it. In order to declare a variable to which properties can be dynamically added, the `Object+` type has to be declared (cf. [Section 4.9.1, “Dynamic”](#)).

### Attributes

`constructor:Object` Returns a reference to the `Object` function that created the instance's prototype.

### Methods

`toString():Object` Returns a string representing the specified object.

`toLocaleString():Object` Returns a string representing the object. This method is meant to be overridden by derived objects for locale-specific purposes.

`valueOf():Object` Returns the primitive value of the specified object.

`hasOwnProperty(prop:String):boolean` Returns a boolean indicating whether an object contains the specified property as a direct property of that object and not inherited through the prototype chain.

`isPrototypeOf(object:Object):boolean` Returns a boolean indication whether the specified object is in the prototype chain of the object this method is called upon.

`propertyIsEnumerable(prop:PropertyKey):boolean` Returns a boolean indicating if the internal ECMAScript `DontEnum` attribute is set.

### Static Methods

`getPrototypeOf(object:Object):Object` Returns the prototype of the specified object.

`create(object:Object, properties:PropertyKey[]):Object` Creates an object with the specified prototype object and properties.

`defineProperty(object:Object, property:PropertyKey, descriptor:PropertyDescriptor):Object` Defines a new property directly on an object or overrides an existing property on an object and returns the object.

`defineProperties(object:Object, properties:PropertyKey[], value:any):Object` Defines multiple properties directly on an object, returning the object.

`seal(object:Object, properties:PropertyKey[]):Object` Seals an object, preventing new properties from being added to it and marking all existing properties as non-configurable. Values of present properties can still be changed as long as they are writable.

`freeze(object:Object):Object` Freezes an object: that is, prevents new properties from being added to it, prevents existing properties from being removed, prevents existing properties or their enumerability, configurability, or writability from being changed. In essence, the object is made effectively immutable. The method returns the object being frozen.

`preventExtensions(object:Object)` Prevents new properties from ever being added to an object (i.e. prevents future extensions to the object).

`isSealed(object:Object):boolean` Determine if an object is sealed.

`static`

`isFrozen(object:Object):boolean` Determine if an object is frozen.

`isExtensible(object:Object):boolean` Determine if an object is extensible (whether it can have new properties added to it).

`keys(object:Object):Array<PropertyKey>` Returns an array of all own enumerable properties found upon a given object in the same order as that provided by a for-in loop (the difference being that a for-in loop enumerates properties in the prototype chain as well).

## 17.2. String

`String` is a global object that may be used to construct `String` instances and is a sub class of `Object`.

### Attributes

`number: length` The length of a string.

## Methods

```

String(thing:Object=)      -
anchor(anchorname:String) Creates an HTML anchor.

big():String            Returns a string in a big font.
blink():String           Returns a string in a blinking string.
bold():String            Returns a string in a bold font.

charAt(index:Number):String Returns the character at a specified position.

charCodeAt(index:Number):Number Returns the Unicode of the character at a specified position.

concat(strings:String...):String Joins two or more strings.

equals(object:Object):Boolean
equalsIgnoreCase(object:Object):Boolean

fromCharCode(num:Any...):String Returns a string created by using the specified sequence of Unicode values.

fixed():String          Returns a string as teletype text.
fontcolor(color):String   Returns a string in a specified color.
fontsize(size):String    Returns a string in a specified size.

indexOf(searchValue, fromIndex:Number=):Number Returns the position of the first occurrence of a specified string value in a string.

italics():String         Returns a string in italic.

lastIndexOf(searchValue, fromIndex:Number=):Number Returns the position of the last occurrence of a specified string value, searching backwards from the specified position in a string.

link(url):String        Returns a string as a hyperlink.

localeCompare(otherString):Number This method returns a number indicating whether a reference string comes before or after or is the same as the given string in sort order.

match(searchValue):String Searches for a specified value in a string.

replace(findString,newString):String Replaces some characters with some other characters in a string.

search(searchValue):Number Searches a string for a specified value.

slice(beginSlice:Number, endSlice:Number=):String Extracts a part of a string and returns the extracted part in a new string.

small():String           Returns a string in a small font.

split(separator, howmany:Number=):Array<String> Splits a string into an array of strings.

strike():String          Returns a string with a strikethrough.

sub():String             Returns a string as subscript.

substr(start:Number,length):String Extracts a specified number of characters in a string, from a start index.

substring(beginIndex:number, endIndex:number):String Extracts the characters in a string between two specified indices.

sup():String             Returns a string as superscript.

toLocaleUpperCase():String Returns a string in lowercase letters.

toString():String         Returns a String value for this object.

toUpperCase():String     Returns a string in uppercase letters.

valueOf():String         Returns the primitive value of a String object.

```

## Static Methods

`String(value:Object=)` Static constructor.

## 17.3. Boolean

`Boolean` does not have a super class.

### Static Methods

```
Boolean(value:Object=):Boolean
```

## 17.4. Number

`Number` does not have a super class.

### 17.4.1. Static Attributes

<code>MAX_VALUE:Number</code>	The largest representable number.
<code>MIN_VALUE:Number</code>	The smallest representable number.
<code>NaN:Number</code>	Special 'not a number' value.
<code>NEGATIVE_INFINITY:Number</code>	Special value representing negative infinity, returned on overflow.
<code>POSITIVE_INFINITY:Number</code>	Special value representing infinity, returned on overflow.

### Methods

<code>toExponential(numberOfDecimals?:number=)</code>	Converts a number object into an exponential notation.
<code>toFixed(numberOfDecimals?:number=)</code>	Formats a number to the specified number of decimals.
<code>toPrecision(numberOfDecimals?:number=)</code>	Converts a number into exponential notation if it has more digits than specified.
<code>valueOf():Number</code>	Returns the primitive value of a Number object.
<code>toString(radix:Number=)</code>	Returns a String value for this object. The <code>toString</code> method parses its first argument and attempts to return a string representation in the specified radix (base).
<code>toLocaleString(locale:String  [String]=undefined, options:{NumberFormatOptions=undefined} runtime-ecma402 )</code>	Returns a locale-specific String value for this object. The <code>toLocaleString</code> accepts two optional arguments. The semantics of these arguments is defined in <a href="#">ECMA-402 (Internationalization API Specification)</a> . In N4JS, the base definition does not define that method, instead Number inherits <code>toLocaleString</code> from Object. The specialized definition is found in the runtime library <code>n4js-runtime-ecma402</code> .

### Static Methods

```
Number(value:Object=):Number
```

## 17.5. Function

`Function` does not have a super class.

### Attributes

<code>prototype:Object</code>	Allows the addition of properties to the instance of the object created by the constructor function.
<code>length:Number</code>	Specifies the number of arguments expected by the function.

### Methods

<code>apply(thisArg,argsArray:Array)</code>	Applies the method of another object in the context of a different object (the calling object); arguments can be passed as an Array object.
<code>call(thisArg,arg... ):Object</code>	Calls (executes) a method of another object in the context of a different object (the calling object); arguments can be passed as they are.
<code>bind(thisArg:Object,arg... ):Function</code>	Creates a new function that, when called, itself calls this function in the context of the provided <code>this</code> value with a given sequence of arguments preceding any provided when the new function was called.

## 17.6. Error

`Error` does not have a super class.

## Attributes

`name:String` Error name.  
`message:String` Error message.

## Static Methods

`Error(message:Object=):Error` Static Constructor.

## 17.7. Array

`Array` is a generic type with the type parameter `E` and does not have a super class.

## Methods

`concat(array...):Array<E>` Joins two or more arrays and returns the result.

`every(callback:Function):Boolean` Tests whether all elements in the array pass the test implemented by the provided function. The callback will be called with 3 arguments (elementValue,elementIndex,traversedArray).

`filter(callback:Function):Array<E>` Creates a new array with all elements that pass the test implemented by the provided function. The callback will be called with 3 arguments (elementValue,elementIndex,traversedArray).

`forEach(callback:Function)` Calls a function for each element in the array. The callback will be called with 3 arguments (elementValue,elementIndex,traversedArray). Optionally with a `thisObject` argument to use as this when executing callback.

`indexOf(searchElement,fromIndex):Number` Returns the first index at which a given element can be found in the array, or -1 if it is not present.

`join(separator=:String):String` Puts all the elements of an array into a string. The elements are separated by a specified delimiter.

`lastIndexOf(searchElement,fromIndex?:Number):Number` Returns the last (nearest) index of an element within the array equal to the specified value. Will return -1 if none are found.

`length():Number` The length returns an integer representing the length of an array.

`map(callback:Function,thisObject?:any):Array<E>` Creates a new array with the results of calling a provided function on every element in this array. The callback will be called with 3 arguments (elementValue,elementIndex,traversedArray). Optionally, with a `thisObject` argument to use as this when executing callback.

`pop():E` Removes and returns the last element of an array.

`push(element...):E` Adds one or more elements to the end of an array and returns the new length.

`reverse():Array<E>` Reverses the order of the elements in an array.

`shift()` Removes and returns the first element of an array.

`slice(start:Number,end?:Number):Array<E>` Returns selected elements from an existing array.

`some(callback:Function,thisObject?:any):Boolean` Tests whether some element in the array passes the test implemented by the provided function. The callback will be called with 3 arguments (elementValue,elementIndex,traversedArray). Optionally, with a `thisObject` argument to use as this when executing callback.

`sort(sortByFunction:Function):Array<E>` Sorts the elements of an array. The function will be called with 2 arguments (a,b).

`splice(index:Number,howMany:Number,element...):Array<E>` Removes and adds new elements to an array. Returns the removed elements as an Array.

`toLocaleString():String`, Returns a String value for Array.

`toString():String`

`unshift(element...):E` Adds one or more elements to the beginning of an array and returns the new length.

## Static Methods

`Array(item:Object...)` Static constructor.

## 17.8. Date

`Date` does not have a super class.

## Static Methods

---

```

Date():Date           Static constructor.

Date(milliseconds:Number) Constructor.

Date(date:Date):Date    Constructor.

Date(dateString:String):Date Constructor.

Date(year:Number,month:Number[,day:Number=,hour:Number=,minute:Number=,second:Number=,millisecond:Number=]):Date

parse(dateString:String):Date Parses a string representation of a date, and returns the number of milliseconds since midnight Jan 1, 1970.

now():Number          Returns the numeric value corresponding to the current time.

UTC(year:Number,month:Number[,day:Number=,hour:Number=,minute:Number=,second:Number=,millisecond:Number=]):Date UTC takes numeric immediate parameters and returns the number of milliseconds between January 1, 1970, 00:00:00, Universal Time and the time you specified.

```

## Methods

<b>getDate():Number</b>	Returns the day of the month from a Date object (from 1-31).
<b>getDay():Number</b>	Returns the day of the week from a Date object (from 0-6).
<b>getFullYear():Number</b>	Returns the year, as a four-digit number.
<b>getHours():Number</b>	Returns the hour of a day (from 0-23).
<b>getMilliseconds():Number</b>	Returns the milliseconds of a Date object (from 0-999).
<b>getMinutes():Number</b>	Returns the minutes of a date (from 0-59).
<b>getMonth():Number</b>	Returns the month from a date (from 0-11).
<b>getSeconds():Number</b>	Returns the seconds of a date (from 0-59).
<b>getTime():Number</b>	Returns the number of milliseconds since midnight Jan 1, 1970.
<b>valueOf():Number</b>	Returns the primitive value of a Date object as a number data type, the number of milliseconds since midnight 01 January, 1970 UTC. This method is functionally equivalent to the getTime method.
<b>getTimezoneOffset():Number</b>	Returns the difference in minutes between local time and Greenwich Mean Time (GMT).
<b>getUTCDate():Number</b>	Returns the day of the month from a date according to Universal Time (from 1-31).
<b>getUTCDay():Number</b>	Returns the day of the week from a date according to Universal Time (from 0-6).
<b>getUTCFullYear():Number</b>	Returns the four-digit year from a date according to Universal Time.
<b>getUTCHours():Number</b>	Returns the hour of a date according to Universal Time (from 0-23).
<b>getUTCMilliseconds():Number</b>	Returns the milliseconds of a date according to Universal Time (from 0-999).
<b>getUTCMinutes():Number</b>	Returns the minutes of a date according to Universal Time (from 0-59).
<b>getUTCMonth():Number</b>	Returns the month from a Date object according to Universal Time (from 0-11).
<b>getUTCSeconds():Number</b>	Returns the seconds of a date according to Universal Time (from 0-59).
<b>getYear():Number</b>	Returns the year as a two-digit or a three/four-digit number, depending on the browser. Use getFullYear() instead!
<b>deprecated</b>	
<b> setDate(day):Number</b>	Sets the day of the month from a Date object (from 1-31).
<b> setFullYear(full year, month=, day=):Number</b>	Sets the year as a four-digit number.
<b> setHours(hours,minutes=,seconds=,milliseconds=):Number</b>	Sets the hours, minutes, seconds, and milliseconds of a date (from 0-23).
<b> setMilliseconds(mills):Number</b>	Sets the milliseconds of a Date object (from 0-999).
<b> setMinutes(minutes,=seconds):Number</b>	Sets the minutes and seconds of a date (from 0-59).
<b> setMonth(month):Number</b>	Sets the month from a date (from 0-11).
<b> directType="Number(month,day=):Number</b>	
<b> setSeconds(seconds,millis):Number</b>	Sets the seconds of a date (from 0-59).
<b> setTime(mills):Number</b>	Sets the number of milliseconds since midnight Jan 1, 1970.
<b> setUTCDate(day):Number</b>	Sets the day of the month from a date according to Universal Time (from 0-6).
<b> setUTCFullYear(fullyear,month=,day=):Number</b>	Sets the day, month, and year from a date according to Universal Time.
<b> setUTCHours(hours,minutes):Number</b>	Sets the hour, minutes, and seconds of a date according to Universal Time (from 0-23).

```

setUTCMilliseconds(mills) Sets the milliseconds of a date according to Universal Time (from 0-999).

setUTCMinutes(minutes, seconds) Sets the minutes of a date according to Universal Time time (from 0-59).

setUTCMonth(month, day=) Sets the month from a Date object according to Universal Time (from 0-11).

setUTCSeconds(seconds, milliseconds) Sets the seconds of a date according to Universal Time (from 0-59).

setYear(year):Number Sets the year, as a two-digit or a three/four-digit number, depending on the browser. Use setFullYear() instead!!

toDateDateString():String Returns the date portion of a Date object in readable form.

toLocaleDateString(locale) Converts a Date object, using locales and options as defined in DateTimeFormat of ECMA-402 (Internationalization API), to a string and returns the date and time portion. The toLocalizedString accepts two optional arguments. The semantics of these arguments is defined in ECMA-402 \(Internationalization API Specification\). The specialized definition is found in the runtime library n4js-runtime-ecma402.  

String | [String]=undefined, options:  

_DateTimeFormatOptions=undefined:  

String

toLocaleString(locales: String | [String]=undefined, options: _DateTimeFormatOptions=undefined):String Converts a Date object, using locales and options as defined in DateTimeFormat of ECMA-402 (Internationalization API), to a string. The toLocalizedString accepts two optional arguments. The semantics of these arguments is defined in ECMA-402 \(Internationalization API Specification\). In N4JS, the base definition does not define that method, instead Date inherits toLocalizedString. The specialized definition is found in the runtime library n4js-runtime-ecma402.  

String

toLocaleTimeString(locale) Converts a Date object, using locales and options as defined in DateTimeFormat of ECMA-402 (Internationalization API), to a string and returns the time portion. The semantics of these arguments is defined in ECMA-402 \(Internationalization API Specification\). The specialized definition is found in the runtime library n4js-runtime-ecma402.  

String | [String]=undefined, options: _DateTimeFormatOptions=undefined:  

String

toString():String Returns a String value for this object.

toTimeString():String Returns the time portion of a Date object in readable form.

toUTCString():String Converts a Date object, according to Universal Time, to a string.

```

## 17.9. Math

`Math` is not instantiable and only provides static properties and methods.

### 17.9.1. Static Attributes

<b>E:Number</b>	Euler's constant and the base of natural logarithms, approximately 2.718.
<b>LN2:Number</b>	Natural logarithm of 2, approximately 0.693.
<b>LN10:Number</b>	Natural logarithm of 10, approximately 2.302.
<b>LOG2E:Number</b>	Base 2 logarithm of E, approximately 1.442.
<b>LOG10E:Number</b>	Base 10 logarithm of E, approximately 0.434.
<b>PI:Number</b>	Ratio of the circumference of a circle to its diameter, approximately 3.14159.
<b>SQRT1_2:Number</b>	Square root of 1/2; equivalently, 1 over the square root of 2, approximately 0.707.
<b>SQRT2:Number</b>	Square root of 2, approximately 1.414.

### Static Methods

<b>abs(x):Number</b>	Returns the absolute value of a number.
<b>acos(x:Number):Number</b>	Returns the arccosine of a number.
<b>asin(x:Number):Number</b>	Returns the arcsine of a number.
<b>atan(x:Number):Number</b>	Returns the arctangent of a number.
<b>atan2(y:Number,x:Number):Number</b>	Returns the arctangent of the quotient of its arguments.
<b>ceil(x):Number</b>	Returns the smallest integer greater than or equal to a number.
<b>cos(x):Number</b>	Returns the arctangent of the quotient of its arguments.

<code>exp(x):Number</code>	Returns Enumber, where number is the argument, and E is Euler's constant (2.718...), the base of the natural logarithm.
<code>floor(x):Number</code>	Returns the largest integer less than or equal to a number.
<code>log(x):Number</code>	Returns the natural logarithm (loge, also ln) of a number.
<code>max(value...):Number</code>	Returns the largest of zero or more numbers.
<code>min(value...):Number</code>	Returns the smallest of zero or more numbers.
<code>pow(base:Number, exponent:Number)</code>	Returns base <sup>exponent</sup> , that is, base <sup>exponent</sup> .
<code>random():Number</code>	Returns a pseudorandom number between 0 and 1.
<code>round(x:Number):Number</code>	Returns the value of a number rounded to the nearest integer.
<code>sin(x:Number):Number</code>	Returns the sine of a number.
<code>sqrt(x:Number):Number</code>	Returns the positive square root of a number.
<code>tan(x:Number):Number</code>	Returns the tangent of a number.

## 17.10. RegExp

`RegExp` does not have a super class.

### Attributes

<code>global:Boolean</code>	Whether to test the regular expression against all possible matches in a string, or only against the first.
<code>ignoreCase:Boolean</code>	Whether to ignore case while attempting a match in a string.
<code>lastIndex:Number</code>	The index at which to start the next match.
<codemultiline:boolean< code=""></codemultiline:boolean<>	Whether or not to search in strings across multiple lines.
<code>source:String</code>	The text of the pattern.

### Methods

<code>exec(str:String):Array</code>	Executes a search for a match in its string parameter.
<code>test(str:String):Boolean</code>	Tests for a match in its string parameter.

## 17.11. JSON

`JSON` is a global object and a subclass of `Object`. Its functionality is provided by two static methods. It is not possible to create new instances of type `JSON`.

### Attributes

The `JSON` object does not define own properties.

### Methods

The `JSON` object does not define own methods.

### Static Methods

The `parse` function parses a JSON text (a JSON-formatted String) and produces an ECMAScript value. The JSON format is a restricted form of ECMAScript literal. JSON objects are realized as ECMAScript objects. JSON arrays are realized as ECMAScript arrays. JSON strings, numbers, booleans, and null are realized as ECMAScript Strings, Numbers, Booleans, and null. For detailed information see [ECMA11a(p.S15.12.2)]

The optional `reviver` parameter is a function that takes two parameters (key and value). It can filter and transform the results. It is called with each of the key/value pairs produced by the `parse` and its return value is used instead of the original value. If it returns what it received, the structure is not modified. If it returns then the property is deleted from the result.

The `stringify` function returns a String in JSON format representing an ECMAScript value. It can take three parameters. The first parameter is required. The `value` parameter is an ECMAScript value which is usually an object or array, although it can also be a String, Boolean, Number or null.

The optional `replacer` parameter is either a function that alters the way objects and arrays are stringified or an array of Strings and Numbers that act as a white list for selecting the object properties that will be stringified.

The optional space parameter is a String or Number that allows the result to have whitespace injected into it to improve human readability.

For detailed information see [[ECMA11a\(p.S15.12.3\)](#)].

# Chapter 18. N4JS Objects

## 18.1. Reflection Model

N4JS provided metadata for reflection (and introspection). This information is available via instances of some reflection model classes (described below) attached to types and (in some cases) functions.

The following class diagrams shows the defined classes. Note that for performance reasons, the actual structure may vary from the model.

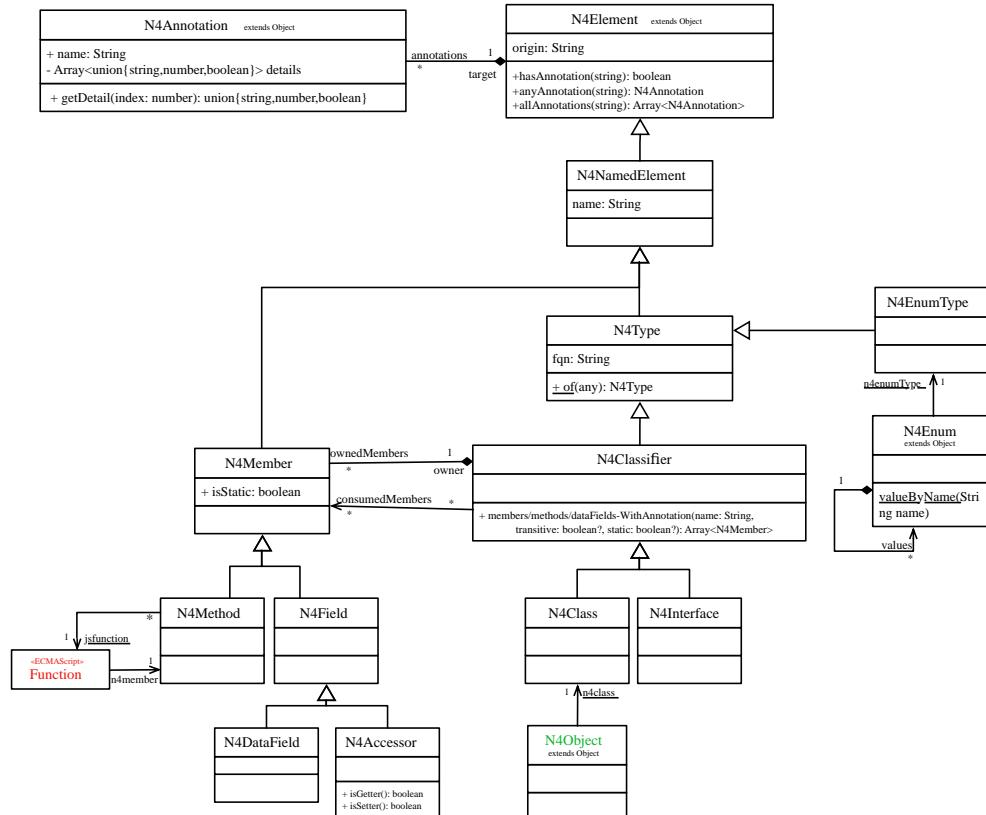


Figure 18.1. N4JS Reflection Classes

Remark: This section is work in progress. The final goal is to provide a metamodel similar to Ecore, but we will only add new features if needed.

```

/**
 * Base class for all N4 reflective elements.
 */
export public abstract class N4Element extends Object {
    /**
     * Annotations attached to this element.
     */
    public Array<N4Annotation> annotations = [];

    /**
     * The origin string formed as <ID-VERSION>, where ID is containing project artefact id, and VERSION is its version
     */
    public String origin;

    /**
     * Returns true if an annotation with the given name is attached to the element.
     */
    public boolean hasAnnotation(string name) {
        return !!this.anyAnnotation(name)
    }

    /**
     * Returns any annotation with the given name attached to the element, or null if no such annotation is found.
     */
    public N4Annotation anyAnnotation(string name) {
        for (var i=this.annotations.length-1; i>=0; i--) {
            var a = this.annotations[i];
        }
    }
}

```

```

        if (a) {
            return a;
        }
    }
    return null;
}
/***
 * Returns all annotations with the given name attached to the element, or an empty array if no such annotations are found.
 */
public Array<N4Annotation> allAnnotations(string name) {
    return this.annotations.filter(function(a){return a.name==name});
}
}

/***
 * Base class for all reflective classes with a name.
 */
export public abstract class N4NamedElement extends N4Element {
    /**
     * The simple name of a named element.
     */
    public string name;
}

/***
 * Base class for all reflective classes describing a type (declaration).
 */
export public abstract class N4Type extends N4NamedElement {
    /**
     * The FQN of the type.
     */
    public string fqn;

    /**
     * Returns true if this N4Class describes an N4-class declaration.
     */
    public boolean get isClass() { return false; }

    /**
     * Returns true if this N4Class describes an N4-interface declaration.
     */
    public boolean get isInterface() { return false; }

    /**
     * Returns true if this N4Class describes an N4-enumeration declaration.
     */
    public boolean get isEnum() { return false; }
}

/***
 * Base class for meta types of classes or interfaces.
 */
export public abstract class N4Classifier extends N4Type {

    /**
     * The N4Class of the super type, may be null if super type is a not an N4Class.
     */
    public N4Class n4superType;

    /**
     * Array of the FQN of all (transitively) implemented interfaces, i.e. interfaces directly implemented by this class, its super
     * * class or interfaces extended by directly implemented interfaces.
     */
    public Array<string> allImplementedInterfaces;

    /**
     * Array of all owned members, that is members defined in the class directly.
     * This field is private as it is an internal detail, members are accessed via methods defined in this class.
     */
    private Array<N4Member> ownedMembers;

    /**
     * Array of all consumed members, that is members mixed into the classifier via interface implementation and default methods.
     * This field is private as it is an internal detail, members are accessed via methods defined in this class.
     */
    private Array<N4Member> consumedMembers;

    /**
     * Only used internally, must not be called by client.
    */
}

```

```

/*
protected constructor(@Spec ~this spec) {}

/**
 * Returns all members defined by this class directly, consumed, and inherited. The boolean flags control which members are
returned.
 *
 * @param consumed if set, consumed members are returned as well (false by default)
 * @param inherited if set, inherited members are returned as well (false by default)
 * @param _static if set, static members are returned, otherwise instance members (false by default).
 * @return array of members, may be empty but never null
 */
public Array<? extends N4Member> members(boolean? consumed, boolean? inherited, boolean? _static) {
    return null; // TODO
}

/**
 * Returns all members defined in this classifier (or inherited) with an annotation
 * of given name attached to it. The boolean flags control which methods are returned.
 *
 * @param name name of annotation to be used as filter
 * @param consumed if set, consumed members are returned as well (false by default)
 * @param inherited if set, inherited members are returned as well (false by default)
 * @param _static if set, static members are returned, otherwise instance members (false by default).
 * @return array of members, may be empty but never null
 */
public Array<? extends N4Member> membersWithAnnotation(string name, boolean? consumed, boolean? inherited, boolean? _static) {
    return null; // TODO
}

/**
 * Returns all data fields defined by this class directly, consumed, and inherited. The boolean flags control which data
fields are returned.
 *
 * @param consumed if set, consumed data fields are returned as well (false by default)
 * @param inherited if set, inherited data fields are returned as well (false by default)
 * @param _static if set, static data fields are returned, otherwise instance members (false by default).
 * @return array of data fields, may be empty but never null
 */
public Array<? extends N4DataField> dataFields(boolean? consumed, boolean? inherited, boolean? _static) {
    return null; // TODO
}

/**
 * Returns all data fields defined in this classifier (or inherited) with an annotation
 * of given name attached to it. The boolean flags control which data fields are returned.
 *
 * @param name name of annotation to be used as filter
 * @param consumed if set, consumed data fields are returned as well (false by default)
 * @param inherited if set, inherited data fields are returned as well (false by default)
 * @param _static if set, static data fields are returned, otherwise instance members (false by default).
 * @return array of data fields, may be empty but never null
 */
public Array<? extends N4DataField> dataFieldsWithAnnotation(string name, boolean? consumed, boolean? inherited, boolean?
_static) {
    return null; // TODO
}

/**
 * Returns all methods defined by this class directly, consumed, and inherited. The boolean flags control which methods are
returned.
 *
 * @param consumed if set, consumed methods are returned as well (false by default)
 * @param inherited if set, inherited methods are returned as well (false by default)
 * @param _static if set, static methods are returned, otherwise instance members (false by default).
 * @return array of methods, may be empty but never null
 */
public Array<? extends N4Method> methods(boolean? consumed, boolean? inherited, boolean? _static) {
    return null; // TODO
}

/**
 * Returns all methods defined in this classifier (or inherited) with an annotation
 * of given name attached to it. The boolean flags control which methods are returned.
 *
 * @param name name of annotation to be used as filter
 * @param consumed if set, consumed methods are returned as well (false by default)
 * @param inherited if set, inherited methods are returned as well (false by default)
 * @param _static if set, static methods are returned, otherwise instance members (false by default).
 * @return array of methods, may be empty but never null
*/

```

```

/*
public Array<? extends N4Method> methodsWithAnnotation(string name, boolean? consumed, boolean? inherited, boolean? _static) {
    return null; // TODO
}

}

/***
 * Meta information of an n4 class.
 */
export @Final public class N4Class extends N4Classifier {

    /**
     * Returns the N4Class instance for a given n4object. This is similar to
     * {@code n4object.constructor.n4type}, however it can also be used in interfaces
     * to get reflective information of the implementor.
     */
    public static N4Class of(N4Object n4object) {
        return n4object.constructor.n4type
    }

    /**
     * Returns true if this N4Class describes an N4-class declaration.
     */
    @Override
    public boolean get isClass() { return true; }
}

/***
 * Meta information of an n4 interface.
 */
export @Final public class N4Interface extends N4Classifier {

    /**
     * Returns true if this N4Class describes an N4-interface declaration.
     */
    @Override
    public boolean get isInterface() { return true; }
}

/***
 * Description of a member, that is a method or field.
 */
export public abstract class N4Member extends N4Element {
    public string name;
}

/***
 * Description of a method.
 */
export @Final public class N4Method extends N4Member {
    public Function jsFunction;
}

/***
 * Description of a simple data field.
 */
export @Final public class N4DataField extends N4Member {
}

/***
 * Description of an accessor, that is a getter or setter.
 */
export @Final public class N4Accessor extends N4Member {
    /**
     * Flag indicating whether accessor is a getter or setter, internal detail.
     */
    private boolean getter;
    /**
     * Returns true if accessor is a getter.
     */
    public boolean isGetter() { return this.getter; }
    /**
     * Returns true if accessor is a setter.
     */
    public boolean isSetter() { return ! this.getter; }
}
*/

```

```

 * Description of an N4Enum
 */
export @Final public class N4EnumType extends N4Type {
    /**
     * Returns true if this N4Clasifier describes an N4-enumeration declaration.
     */
    @Override public boolean get isEnum() { return true; }
    /**
     * Returns the N4EnumType instance for a given enum literal. This is similar to
     * {@code n4enum.constructor.n4type}.
     */
    public static N4EnumType of(N4Enum n4enum) {
        return n4enum.constructor.n4type
    }
}

/**
 * Base class for all enumeration, literals are assumed to be static constant fields of concrete subclasses.
 */
export public abstract class N4Enum extends Object {

    /**
     * Returns the name of a concrete literal
     */
    public abstract string get name();

    /**
     * Returns the value of a concrete literal. If no value is
     * explicitly set, it is similar to the name.
     */
    public abstract string get value();

    /**
     * Returns a string representation of a concrete literal, it returns
     * the same result as value()
     */
    @Override public string toString() { return this.value }

    /**
     * Returns the enum class object of this enum literal for reflection.
     * The very same meta class object can be retrieved from the enumeration type directly.
     */
    public abstract N4Enum get n4Enum()

    /**
     * Natively overridden by concrete enums.
     */
    public static Array<? extends N4Enum> get values() { return null; }

    /**
     * Natively overridden by concrete enums.
     */
    public static N4Enum valueByName(string name) { return null; }

    /**
     * Returns the meta class object of this class for reflection.
     * The very same meta class object can be retrieved from an instance by calling
     * <code>instance.constructor.n4type</code>
     */
    public static N4EnumType get n4type() { return null; }
}

/**
 * Annotation with value.
 */
export @Final public class N4Annotation extends Object {
    public string name;
    public union{string,number} value;
}

/**
 * The base class for all instances of n4 classes.
 */
export public class N4Object {
    /**
     * Returns the meta class object of this class for reflection.
     * The very same meta class object can be retrieved from an instance by calling
     * <code>instance.constructor.n4type</code>
     */
    // defined in types model, added by $makeClass:
}

```

```
// public static N4Class get n4type() { return null; }  
}
```

## 18.2. Error Types

N4JS provides additional Error types as subtypes of `Error`.

### 18.2.1. N4ApiNotImplemented

Considering API definitions and concrete implementations of those APIs the error `N4-Api-Not-Implemented-Error` is introduced to specifically report missing implementations. Instances of this error type are inserted internally during the transpilation of API-implementing projects. Whenever a difference to the API in form of a missing implementation is encountered, the transpiler will insert stub-code throwing an instance of `N4-Api-Not-Implemented-Error`.

API-testing projects can catch those errors and act accordingly. This enables tracking of completeness of implementations by counting the occasions an `N4-Api-NotImplemented-Error` was encountered.

```
/**  
 * Error type reporting a not implemented situation.  
 */  
public class N4ApiNotImplementedError extends Error { }
```

# Appendix A. Acronyms

<b>CDep</b>	Compile-Time Dependency	<b>RDep</b>	Run-Time Dependency
<b>LDep</b>	Load-Time Dependency	<b>IDep</b>	Initialization-Time Dependency
<b>EDep</b>	Execution-Time Dependency	<b>ANTLR</b>	ANother Tool for Language Recognition
<b>API</b>	Application Programming Interface	<b>AST</b>	Abstract Syntax Tree
<b>ASI</b>	Automatic Semicolon Insertion	<b>BNF</b>	Backus-Naur Form
<b>CLI</b>	Command Line Interface (including a headless compiler and runner.)	<b>DI</b>	Dependency Injection
<b>DIC</b>	DI Component	<b>DOM</b>	Document Object Model
<b>DSL</b>	Domain Specific Language	<b>EBNF</b>	Extended Backus-Naur Form
<b>EMF</b>	Eclipse Modeling Framework	<b>FQN</b>	Fully Qualified Name
<b>GLB</b>	Greatest Lower Bound, also known as infimum	<b>GCST</b>	Greatest Common Sub Type, also known as meet
<b>IDE</b>	Integrated Development Environment	<b>IDL</b>	Interface Definition Language
<b>LSP</b>	Liskov Substitution Principle [ <a href="#">Martin96b</a> ]	<b>LUB</b>	Least Upper Bound, also known as supremum
<b>LCST</b>	Least Common Super Type, also known as join	<b>N4JS</b>	N4JS for JavaScript
<b>N4JSED</b>	N4JS Environment Definition	<b>N4JSIDE</b>	N4JS Integrated Development Environment (Eclipse-based IDE for all N4JS related languages and projects)
<b>VM</b>	Virtual Machine	<b>XML</b>	Extensible Markup Language
<b>XSLT / XSL</b>	XSL Transformations	<b>XSL</b>	Extensible Stylesheet Language
<b>WYSIWYG</b>	What You See Is What You Get	<b>WLOG</b>	Without loss of generality



# Appendix B. Bibliography

- ECMA. (2015). *ECMAScript 2015 Language Specification*. Retrieved from <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>
- (2011). *ECMAScript Language Specification*. Retrieved from <http://www.ecma-international.org/publications/standards/Ecma-262.htm>
- Pierce, Benjamin C.. (2002). *Types and Programming Languages*.
- Gosling, James and Joy, Bill and Steele, Guy and Bracha, Gilad and Buckley, Alex. (2015). *The Java Language Specification. Java SE 8 Edition*. Retrieved from <http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>
- Gosling, James and Joy, Bill and Steele, Guy and Bracha, Gilad and Buckley, Alex. (2012). *The Java Language Specification. Java SE 7 Edition*. Retrieved from <http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf>
- Nielson, Flemming and Nielson, HanneRiis. (1999). *Type and Effect Systems*. Retrieved from [http://dx.doi.org/10.1007/3-540-48092-7\\_6](http://dx.doi.org/10.1007/3-540-48092-7_6)
- Crary, Karl and Weirich, Stephanie and Morrisett, Greg. (2002). *Intensional polymorphism in type-erasure semantics*. Retrieved from [http://journals.cambridge.org/article\\_S0956796801004282](http://journals.cambridge.org/article_S0956796801004282)
- Igarashi, Atsushi and Pierce, Benjamin C. and Wadler, Philip. (2001). *Featherweight Java: a minimal core calculus for Java and GJ*.
- Torgersen, Mads and Ernst, Erik and Hansen, Christian Plesner. (2005). *Wild FJ*. Retrieved from <http://homepages.inf.ed.ac.uk/wadler/fool/program/14.html>
- Cameron, Nicholas and Drossopoulou, Sophia and Ernst, Erik. (2008). *A Model for Java with Wildcards*. Retrieved from [http://dx.doi.org/10.1007/978-3-540-70592-5\\_2](http://dx.doi.org/10.1007/978-3-540-70592-5_2)
- Cameron, Nicholas. (2009). *Existential Types for Variance—Java Wildcards and Ownership Types*. Retrieved from [http://www.doc.ic.ac.uk/~ncameron/papers/cameron\\_thesis.pdf](http://www.doc.ic.ac.uk/~ncameron/papers/cameron_thesis.pdf)
- Summers, Alexander J. and Cameron, Nicholas and Dezani-Ciancaglini, Mariangiola and Drossopoulou, Sophia. (2010). *Towards a semantic model for Java wildcards*.
- Wehr, Stefan and Thiemann, Peter. (2008). *Subtyping Existential Types*. Retrieved from [http://www.informatik.uni-freiburg.de/~wehr/publications/Wehr\\_Subtyping\\_existential\\_types.pdf](http://www.informatik.uni-freiburg.de/~wehr/publications/Wehr_Subtyping_existential_types.pdf)
- Igarashi, Atsushi and Nagira, Hideshi. (2007). *Union Types for Object-Oriented Programming*. Retrieved from [http://www.jot.fm/issues/issue\\_2007\\_02/article3/](http://www.jot.fm/issues/issue_2007_02/article3/)
- King, Gavin. (2013). *The Ceylon Language*. Red Hat, Inc.. Retrieved from <http://ceylon-lang.org/documentation/1.0/spec/pdf/ceylon-language-specification.pdf>
- Laurent, Olivier. (2012). *Intersection types with subtyping by means of cut elimination*.
- Steen, Hallvard and Aubourg, Julian and van Kesteren, Anne and Song, Jungkee. (2014). *XMLHttpRequest Level 1*.
- Dart Team. (2013). *Dart Programming Language Specification*. Retrieved from <http://www.dartlang.org/docs/spec/latest/dart-language-specification.pdf>
- OMG. (2014). *Interface Definition Language*. Object Management Group. Retrieved from <http://www.omg.org/cgi-bin/doc?formal/2014-03-01.pdf>
- W3C. (2012). *Web IDL Specification*. Retrieved from <http://www.w3.org/TR/2012/CR-WebIDL-20120419/>
- Kuizinas, Gajus. (2014). *The Definitive Guide to the JavaScript Generators*. Retrieved from <http://gajus.com/blog/2/the-definitive-guide-to-the-javascript-generators>
- West, Mike. (2006). *Scope in JavaScript*. Retrieved from [http://www.digital-web.com/articles/scope\\_in\\_javascript/](http://www.digital-web.com/articles/scope_in_javascript/)
- Fowler, Martin. (2004). *Inversion of Control Containers and the Dependency Injection pattern*. Retrieved from <http://martinfowler.com/articles/injection.html>
- Prasanna, Dhanji R. (2009). *Dependency Injection: Design Patterns Using Spring and Guice*.

---

Zhu, He and Jagannathan, Suresh. (2013). *Compositional and Lightweight Dependent Type Inference for ML*. Retrieved from [http://dx.doi.org/10.1007/978-3-642-35873-9\\_19](http://dx.doi.org/10.1007/978-3-642-35873-9_19)

Hudli, Shrinidhi R. and Hudli, Raghu V.. (2013). *A Verification Strategy for Dependency Injection*.

Lesiecki, Nicholas. (2008). *Dependency injection with Guice*. Retrieved from <http://www.ibm.com/developerworks/library/j-guice/>

Betts, Dominic and Melnik, Grigori and Simonazzi, Fernando and Subramanian, Mani. (2013). *Dependency Injection with Unity*.

Knol, Alex. (2013). *Dependency injection with AngularJS*.

Dagger, Project Website. Retrieved from <http://square.github.io/dagger/>

(2012). *ECMAScript Internationalization API Specification*. Retrieved from <http://www.ecma-international.org/publications/standards/Ecma-402.htm>

Martin, Robert C. (1996). *The Liskov Substitution Principle*. Retrieved from <http://www.objectmentor.com/publications/lsp.pdf>